Introduction to Computational Linguistics

Marcus Kracht
Department of Linguistics, UCLA
3125 Campbell Hall
450 Hilgard Avenue
Los Angeles, CA 90095–1543
kracht@humnet.ucla.edu

1 General Remarks

This lecture is an introduction to computational linguistics. As such it is also an introduction to the use of the computer in general. This means that the course will not only teach theoretical methods but also practical skills, namely programming. In principle, this course does not require any knowledge of either mathematics or programming, but it does require a level of sophistication that is acquired either by programming or by doing some mathematics (for example, an introduction to mathematical linguistics).

The course uses the language OCaML for programming. It is however not an introduction into OCaML itself. It offers no comprehensive account of the language and its capabilities but focuses instead on the way it can be used in practical computations. OCaML can be downloaded from the official site at

```
http://caml.inria.fr/
```

together with a reference manual and a lot of other useful material. (The latest release is version number 3.09.1 as of January 2006.) In particular, there is a translation of an introduction into programming with OCaML which is still not published and can therefore be downloaded from that site (see under OCaML Light). The book is not needed, but helpful. I will assume that everyone has

a version of OCaML installed on his or her computer and has a version of the manual. If help is needed in installing the program I shall be of assistance.

The choice of OCaML perhaps needs comment, since it is not so widely known. In the last twenty years the most popular language in computational linguistics (at least as far as introductions was concerned) was certainly Prolog. However, its lack of imperative features make its runtime performance rather bad for the problems that we shall look at (though this keeps changing, too). Much of actual programming nowadays takes place in C or C++, but these two are not so easy to understand, and moreover take a lot of preparation. OCaML actually is rather fast, if speed is a concern. However, its main features that interest us here are:

- ☞ It is completely typed. This is particularly interesting in a linguistics course, since it offers a view into a completely typed universe. We shall actually begin by explaining this aspect of OCaML.
- ☞ It is completely functional. In contrast to imperative languages, functional languages require more explicitness in the way recursion is handled. Rather than saying that something needs to be done, one has to say how it is done.
- ☞ It is object oriented. This aspect will not be so important at earlier stages, but turns into a plus once more complex applications are looked at.

2 Practical Remarks Concerning OCaML

When you have installed OCaML, you can invoke the program in two ways:

① In Windows by clicking on the icon for the program. This will open an interactive window much like the xterminal in `Unix`.

② by typing `ocaml` after the prompt in a terminal window.

Either way you have a window with an interactive shell that opens by declaring what version is running, like this:

(1)
```
Objective Caml version 3.09.1
```

It then starts the interactive session by giving you a command prompt: #. It is then waiting for you to type in a command. For example

(2)
```
# 4+5;;
```

It will execute this and return to you as follows:

(3)
```
- :  int = 9
#
```

So, it gives you an answer (the number 9), which is always accompanied by some indication of the type of the object. (Often you find that you get only the type information, see below for reasons.) After that, OCaML gives you back the prompt. Notice that your line must end in ;; (after which you have to hit < return >, of course). This will cause OCaML to parse it as a complete expression and it will try to execute it. Here is another example.

(4)
```
# let a = 'q';;
```

In this case, the command says that it should assign the character q to the object with name a. Here is what OCaML answers:

(5)
```
val a :  char = 'q'
```

Then it will give you the prompt again. Notice that OCaML answers without using the prompt, #. This is how you can tell your input from OCaMLs answers. Suppose you type

(6)
```
# Char.code 'a';;
```

Then OCaML will answer

(7)
```
- :  int = 113
```

Then it will give you the prompt again. At a certain point this way of using OCaML turns out to be tedious. For even though one can type in entire programs this way, there is no way to correct mistakes once a line has been entered. Therefore, the following is advisable to do.

First, you need to use an editor (I recommend to use either emacs or vim; vim runs on all platforms, and it can be downloaded and installed with no charge

and with a mouseclick). Editors that Windows provides by default are not good. Either they do not let you edit a raw text file (like Word or the like) or they do not let you choose where to store data (like Notepad). So, go for an independent editor, install it (you can get help on this too from us).

Using the editor, open a file < myfile > `.ml` (it has to have the extension `.ml`). Type in your program or whatever part of it you want to store separately. Then, whenever you want OCaML to use that program, type after the prompt:

(8) `# #use "<myfile>.ml";;`

(The line will contain two #, one is the prompt of OCaML, which obviously you do not enter, and the other is from you! And no space in between the # and the word `use`.) This will cause OCaML to look at the content of the file as if it had been entered from the command line. OCaML will process it, and return a result, or, which happens very frequently in the beginning, some error message, upon which you have to go back to your file, edit it, write it, and load it again. You may reuse the program, this will just overwrite the assignments you have made earlier. Notice that the file name must be enclosed in quotes. This is because OCaML expects a string as input for `#use` and it will interpret the string as a filename. Thus, the extension, even though technically superfluous, has to be put in there as well.

You may have stored your program in several places, because you may want to keep certain parts separate. You can load them independently, but only insofar as the dependencies are respected. If you load a program that calls functions that have not been defined, OCaML will tell you so. Therefore, make sure you load the files in the correct order. Moreover, if you make changes and load a file again, you may have to reload every file that depends on it (you will see why if you try...).

Of course there are higher level tools available, but this technique is rather effective and fast enough for the beginning practice.

Notice that OCaML goes through your program three times. First, it parses the program syntactically. At this stage it will only tell you if the program is incorrect syntactically. Everything is left associative, as we will discuss below. The second time OCaML will check the types. If there is a type mismatch, it returns to you and tells you so. After it is done with the types it is ready to execute the program. Here, too, it may hit type mismatches (because some of them are not apparent at first inspection), and, more often, run time errors such as accessing an item that

isn't there (the most frequent mistake).

3 Welcome To The Typed Universe

In OCaML, every expression is typed. There are certain basic types, but types can also be created. The type is named by a string. The following are inbuilt types (the list is not complete, you find a description in the user manual):

(9)

character	`char`
string	`string`
integer	`int`
boolean	`bool`
float	`float`

There are conventions to communicate a token of a given type. Characters must be enclosed in single quotes. `'a'`, for example refers to the character `a`. Strings must be enclosed in double quotes: `"mail"` denotes the string `mail`. The distinction matters: `"a"` is a string, containing the single letter `a`. Although identical in appearance to the character `a`, they must be kept distinct. The next type, integer is typed in as such, for example `10763`. Notice that `"10763"` is a string, not a number! Booleans have two values, typed in as follows: `true` and `false`. Finally, float is the type of floating point numbers. They are distinct from integers, although they may have identical values. For example, `1.0` is a number of type float, `1` is a number of type int. Type in `1.0 = 1;;` and OCaML will respond with an error message:

(10)
```
# 1.0 = 1;;
This expression has type int but is here used with type
    float
```

And it will underline the offending item. It parses expressions from left to right and stops at the first point where things do not match. First it sees a number of type float and expects that to the right of the equation sign there also is a number of type float. When it reads the integer it complains. For things can only be equal via = if they have identical type.

The typing is very strict. Whenever something is declared to be of certain type, it can only be used with operations that apply to things of that type. Moreover,

OCaML immediately sets out to type the expression you have given. For example, you wish to define a function which you name `f`, and which adds a number to itself. This is what happens:

(11)
```
# let f x = x + x;;
val f :  int -> int = <fun>
```

This says that `f` is a function from integers to integers. OCaML knows this because + is only defined on integers. Addition on float is `+.`:

(12)
```
# let g x = x +.  x;;
val g :  float -> float = <fun>
```

There exist type variables, which are `'a`, `'b` and so on. If OCaML cannot infer the type, or if the function is polymorphic (it can be used on more than one type) it writes `'a` in place of a specific type.

(13)
```
# let iden x = x;;
val iden :  'a -> 'a = <fun>
```

There exists an option to make subtypes inherit properties from the supertype, but OCaML needs to be explicitly told to do that. It will not do *any* type conversion by itself.

Let us say something about types in general. Types are simply terms of a particular signature. For example, a very popular signature in Montague grammar consists in just two so-called *type constructors*, $\rightarrow$ and $\bullet$. A **type constructor** is something that makes new types from old types; both $\rightarrow$ and $\bullet$ are binary type constructors. They require two types each. In addition to type constructors we need basic types. (There are also unary type constructors, so this is only an example.) The full definition is as follows.

Definition 1 (Types) *Let B be a set, the set of **basic types**. Then $\mathrm{Typ}_{\rightarrow,\bullet}(B)$, the **set of types** over B, with **type constructors** $\rightarrow$ and $\bullet$, is the smallest set such that*

- $B \subseteq \mathrm{Typ}_{\rightarrow,\bullet}(B)$, *and*
- *if* $\alpha, \beta \in \mathrm{Typ}_{\rightarrow,\bullet}(B)$ *then also* $\alpha \rightarrow \beta, \alpha \bullet \beta \in \mathrm{Typ}_{\rightarrow,\bullet}(B)$.

Each type is interpreted by particular elements. Typically, the elements of different types are different from each other. (This is when no type conversion occurs. But see below.) If x is of type α and y is of type $\beta \neq \alpha$ then x is distinct from y. Here is how it is done. We interpret each type for $b \in B$ by a set M_b in such a way that $M_a \cap M_b = \emptyset$ if $a \neq b$. Notice, for example, that OCaML has a basic type `char` and a basic type `string`, which get interpreted by the set of ASCII–characters, and strings of ASCII–characters, respectively. We may construe strings over characters as function from numbers to characters. Then, even though a single character is thought of in the same way as the string of length 1 with that character, they are now physically distinct. The string `k`, for example, is the function $f : \{0\} \to M_{\texttt{char}}$ such that $f(0) =$ `k`. OCaML makes sure you respect the difference by displaying the string as `"k"` and the character as `'k'`. The quotes are not part of the object, they tell you what its type is.

Given two types α and β we can form the type of functions from α–objects to β–objects. These functions have type $\alpha \to \beta$. Hence, we say that

(14) $M_{\alpha\to\beta} = \{f : f \text{ is a function from } M_\alpha \text{ to } M_\beta\}$

For example, we can define a function f by saying that $f(x) = 2x + 3$. The way to do this in OCaML is by issuing

(15)
```
# let f x = (2 * x) + 3;;
val f :  int -> int = <fun>
```

Now we understand better what OCaML is telling us. It says that the value of the symbol `f` is of type `int -> int` because it maps integers to integers, and that it is a function. Notice that OCaML inferred the type from the definition of the function. We can take a look how it does that.

First, if x is an element of type $\alpha \to \beta$ and y is an element of type α, then x is a function that takes y as its argument. In this situation the string for x followed by the string for y (but separated by a blank) is well–formed for OCaML, and it is of type β. OCaML allows you to enclose the string in brackets, and sometimes it is even necessary to do so. Hence, `f 43` is well–formed, as is `(f 43)` or even `(f (43))`. Type that in and OCaML answers:

(16)
```
# f 43;;
- :  int = 89
```

The result is an integer, and its value is 89. You can give the function `f` any term that evaluates to an integer. It may contain function symbols, but they must be defined already. Now, the functions * and + are actually predefined. You can look them up in the manual. It tells you that their type is `int -> int -> int`. Thus, they are functions from integers to functions from integers to integers. Hence `(2 * x)` is an integer, because `x` is an integer and `2` is. Likewise, `(2 * x) + 3` is an integer.

Likewise, if α and β are types, so is $\alpha \bullet \beta$. This is the **product type**. It contains the set of pairs $\langle x, y \rangle$ such that x is type α and y of type β:

(17) $$M_{\alpha \bullet \beta} = M_\alpha \times M_\beta = \{\langle x, y \rangle : x \in M_\alpha, y \in M_\beta\}$$

Like function types, product types do not have to be officially created to be used. OCaML's own notation is $*$ in place of $\bullet$, everything else is the same. Type in, for example, `('a', 7)` and this will be your response:

(18)
```
- :  char * int = ('a', 7);;
```

This means that OCaML has understood that your object is composed of two parts, the left hand part, which is a character, and the right hand part, which is a number.

It is possible to explicitly define such a type. This comes in the form of a type declaration:

(19)
```
# type prod = int * int;;
type prod = int * int
```

The declaration just binds the string `prod` to the type of pairs of integers, called the product type `int * int` by OCaML. Since there is nothing more to say, OCaML just repeats what you have said. It means it has understood you and `prod` is now reserved for pairs of integers. However, such explicit definition hardly makes sense for types that can be inferred anyway. For if you issue `let h = (3,4)` then OCaML will respond `val h :  int * int = (3,4)`, saying that `h` has the product type. Notice that OCaML will not say that `h` is of type `prod`, even if that follows from the definition.

There is often a need to access the first and second components of a pair. For that there exist functions `fst` and `snd`. These functions are polymorphic. For every product type $\alpha \bullet \beta$, they are defined and `fst` maps the element into its first component, and `snd` onto its second component.

An alternative to the pair constructor is the constructor for **records**. The latter is very useful. It is most instructive to look at a particular example:

(20)
```
type car = {brand :  string; vintage :  int;
    used :  bool};;
```

This defines a type structure called `car`, which has three components: a brand, a vintage and a usedness value. On the face of it, a record can be replicaed with products. However, notice that records can have any number of entries, while a pair must consist of exactly two. So, the record type `car` can be replicated by either `string * (int * bool)` or by `(string * int) * bool`.

But there are also other differences. One is that the order in which you give the arguments is irrelevant. The other is that the names of the projections can be defined by yourself. For example, you can declare that `mycar` is a car, by issuing

(21)
```
# let mycar = {brand = "honda"; vintage = 1977;}
    used = false};;
```

This will bind `mycar` to the element of type `car`. Moreover, the expression `mycar.brand` has the value `honda`. (To communicate that, it has to be enclosed in quotes, of course. Otherwise it is mistaken for an identifier.) It is not legal to omit the specification of some of the values. Try this, for example:

(22)
```
# let mycar = {brand = "honda"};;
Some record field labels are undefined:  vintage used;;
```

This is not a warning: the object named ”mycar” has not been defined.

(23)
```
# mycar.brand;;
Unbound value mycar
```

If you look carefully, you will see that OCaML has many more type constructors, some of which are quite intricate. One is the disjunction, written `|`. We shall use the more suggestive $\cup$. We have

(24) $M_{\alpha\cup\beta} = M_\alpha \cup M_\beta$

Notice that by definition objects of type α are also objects of type $M_{\alpha\cup\beta}$. Hence what we said before about types keeping everything distinct is not accurate. In

fact, what is closer to the truth is that types offer a classifying system for objects. Each type comes with a range of possible objects that fall under it. An object x is of type α, in symbols $x : \alpha$, if and only if $x \in M_\alpha$. The interpretation of basic types must be given, the interpretation of complex types is inferred from the rules, so far (**??**), (**??**) and (**??**). The basic types are disjoint by definition.

There is another constructor, the list constructor. Given a type, it returns the type of lists over that type. You may think of them as sequences from the numbers $0, 1, \ldots, n-1$ to members of α. However, notice that this interpretation makes them look like functions from integers to M_α. We could think of them as members of `int -> `α. However, this is not the proper way to think of them. OCaML keeps lists as distinct objects, and it calls the type constructor `list`. It is a postfix operator (it is put after its argument). You can check this by issuing

(25)
```
# type u = int list;;
type int list
```

This binds the string `u` to the type of lists of integers. As with pairs, there are ways to handle lists. OCaML has inbuilt functions to do this. Lists are communicated using `[`, `;` and `]`. For example, `[3;2;4]` is a list of integers, whose first element (called **head**) is 3, whose second element is 2, and whose third and last element is 4. `['a';'g';'c']` is a list of characters, and it is not astring, while `["mouse"]` is a list containing just a single element, the string `mouse`. There is a constant, `[ ]`, which denotes the empty list. The double colon denotes the function that appends an element to a list: so `3::[4;5]` equals the list `[3;4;5]`. List concatenation is denoted by `@`. The element to the left of the double colon will be the new head of the list, the list to the right the so–called **tail**. You cannot easily mix elements. Try, for example, typing

(26)
```
let h = [3; "st"; 'a'];;
This expression has type string but is here used with
    type int
```

Namely, OCaML tries to type expression. To do this, it opens a list beginning with `3`, so it thinks this is a list of integers. The next item is a string, so no match and OCaML complains. This is the way OCaML looks at it. (It could form the disjunctive type `all = int | char | string` and then take the object to be a list of `all` objects. But it does *not* do that. It can be done somehow, but then OCaML needs to be taken by the hand.)

4 Function Definitions

Functions can be defined in OCaML in various ways. The first is to say explicitly what it does, as in the following case.

(27) `# let appen x = x^'a';;`

This function takes a string and appends the letter `a`. Notice that OCaML interprets the first identifier after `let` as the name of the function to be defined and treats the remaining ones as arguments to be consumed. So they are bound variables.

There is another way to achieve the same, using a constructor much like the λ-abstractor. For example,

(28) `# let app = (fun x -> x^ 'a';;`

Notice that to the left the variable is no longer present. The second method has the advantage that a function like `app` can be used without having to give it a name. You can replace `app` wherever it occurs by `(fun x -> x^ 'a'`. The choice between the methods is basically one of taste and conciseness.

The next possibility is to use a definition by cases. One method is to use `if`$\cdots$`then`$\cdots$`else`, which can be iterated as often as necessary. OCaML provides another method, namely the construct `match`$\cdots$`with`. You can use it to match objects with variables, pairs with pairs of variables, and list with and tail (iterated to any degree needed).

Recursion is the most fundamental tool to defining a function. If you do logic, one of the first things you get told is that just about any function on the integers can be defined by recursion, in fact by what is known as *primitive recursion*, from only one function: the successor function! Abstractly, to define a function by recursion is to do the following. Suppose that $f(x)$ is a function on one argument. Then you have to say what $f(0)$, and second, you have to say what $f(n+1)$ is on the basis of the value $f(n)$. Here is an example. The function that adds x to y, written $x + y$, can be defined by recursion as follows.

(29) $0 + y := y; (n + 1) + y := (n + y) + 1$

At first it is hard to see that this is indeed a definition by recursion. So, let us write 's' for the successor function; then the clauses are

(30) $0 + y := y; s(n) + y := s(n + y)$

Indeed, to know the value of $f(0) = 0 + y$ you just look up y. The value of $f(n + 1) := (s(n)) + y$ is defined using the value $f(n) = n + y$. Just take that value and add 1 (that is, form the successor).

The principle of recursion is widespread and can be used in other circumstances as well. For example, a function can be defined over strings by recursion. What you have to do is the following: you say what the value is of the empty string, and then you say what the value is of $f(\vec{x}^\frown a)$ on the basis of $f(\vec{x})$, where $\vec{x}$ is a string variable and a a letter. Likewise, functions on lists can be defined by recursion.

Now, in OCaML there is no type of natural numbers, just that of integers. Still, definitions by recursion are very important. In defining a function by recursion you have to tell OCaML that you intend such a definition rather than a direct definition. For a definition by recursion means that the function calls itself during evaluation. For example, if you want to define exponentiation for numbers, you can do the following:

(31)
```
let rec expt n x
  if x = 0 then 1 else
  n * (expt n (x-1));;
```

Type this and then say `expt 2 4` and OCaML replies:

(32)
```
- :  int = 16
```

As usual, recursions need to be grounded, otherwise the program keeps calling itself over and over again. If the definition was just given for natural numbers, there would be no problem. But as we have said, there is no such type, just the type of integeres. This lets you evaluate the function on arguments where it is not grounded. In the present example negative numbers let the recursion loop forever. This is why the above definition is not good. Negative arguments are accepted, but never terminate (because the value on which the recursion is based is further removed from the base, here 0). The evaluation of `expt 2 -3` never terminates. Here is what happens:

(33)
```
# expt 2 (-3);;
Stack overflow during evaluation (looping recursion?)
```

One way to prevent this is to check whether the second argument is negative, and then rejecting the input. It is however good style to add a diagnostic so that you

can find out why the program rejects the input. To do this, OCaML lets you define so–called exceptions. This is done as follows.

(34)
```
exception Input_is_negative;;
```

Exceptions always succeed, and they succeed by OCaML typing their name. So, we improve our definition as follows.

(35)
```
let rec expt n x
  if x < 0 then raise Input_is_negative else
  if x = 0 then 1 else
  n * (expt n (x-1))
;;
```

Now look at the following dialog:

(36)
```
# expt 2 (-3)
Exception:  Input_is_negative
```

Thus, OCaML quits the computation and raises the exception that you have defined (there are also inbuilt exceptions). In this way you can make every definition total.

Recursive definitions of functions on strings or lists are automatically grounded if you include a basic clause for the value on the empty string or the empty list, respectively. It is a common source of error to omit such a clause. Beware! However, recursion can be done in many different ways. For strings, for example, it does not need to be character by character. You can define a function that takes away two characters at a time, or even more. However, you do need to make sure that the recursion ends somewhere. Otherwise you will not get a reply.

5 Modules

Modules are basically convenient ways to package the programs. When the program gets larger and larger, it is sometimes difficult to keep track of all the names and identifiers. Moreover, one may want to identify parts of the programs that

can be used elsewhere. The way to do this is to use a module. The syntax is as follows:

(37)
```
module name-of-module =
  struct
any sequence of definitions
  end;;
```

For example, suppose we have defined a module called `Seminar`. It contains definitions of objects and functions and whatever we like, for example a function `next_speaker`. In order to use this function outside of the module, we have to write `Seminar.next_speaker`. It is useful at this point to draw attention to the naming conventions (see the reference manual). Some names must start with a lower-case letter (names of functions, objects and so on), while others must start with an uppercase-letter. The latter are names of exceptions (see below), and names of modules. So, the module may not be called `seminar`. In the manual you find a long list of modules that you can use, for example the module `List`. You may look at the source code of this module by looking up where the libraries are stored. On my PC they sit in

```
usr/local/lib/ocaml
```

There you find a file called `list.ml` (yes, the file is named with a lower case letter!) and a file `list.mli`. The first is the plain source code for the list module. You can look at it and learn how to code the functions. The file `list.mli` contains the type interface. More on that later. The first function that you see defined is `length`. This function you can apply to a list and get its length. The way you call it is however by `List.length`, since it sits inside the module `List`. However, you will search in vain for an explicit definition for such a module. Instead OCaML creates this module from the file `list.ml`. If you compile this file separately, its content will be available for you in the form of a module whose name is obtained from the filename by raising the initial letter to upper case (whence the file name `list.ml` as opposed to the module name `List`).

Lets look at the code. The way this is done is interesting.

(38)
```
let rec length_aux len = function
    [] -> len
  | a::l -> length_aux (len + 1) l
```

```
let length l = length_aux 0 l
```

First, the function is defined via an auxiliary function. You expect therefore to be able to use a function `List.length_aux`. But you cannot. The answer to the puzzle is provided by the file `list.mli`. Apart form the commentary (which OCaML ignores anyway) it only provides a list of functions and their types. This is the type interface. It tells OCaML which of the functions defined in `list.ml` are public. By default all functions are public (for example, if you did not make a type interface file). If a type interface exists, only the listed functions can be used outside of the module. The second surprise is the definition of the function `length_aux`. It uses only one argument, but at one occasion uses two! The answer is that a function definition of the form `let f x =` specifies that x is the first argument of f, but the resulting type of the function may be a function and so take further arguments. Thus, unless you have to, you need not mention all arguments. However, dropping arguments must proceed from the rightmost boundary. For example, suppose we have defined the function `exp x y`, which calculates x^y. The following function will yield [5; 25; 125] for the list [1;2;3]:

(39)
```
let lexp l = List.map (exp 5) l
```

You may even define this function as follows:

(40)
```
let lexp = List.map (exp 5)
```

However, if you want to define a function that raises every member of the list to the fifth power, this requires more thought. For then we must abstract from the innermost argument position. Here is a solution:

(41)
```
let rexp = List.map (fun y -> exp y 5)
```

A more abstract version is this:

(42)
```
let switch f x y = f y x;;
let lexp = List.map ((switch exp) 5)
```

The function `switch` switches the order of the function (in fact any function). The so defined reversal can be applied in the same way as the original, with arguments now in reverse order.

I have said above that a module also has a signature. You may explicitly define signatures in the following way.

(43)
```
module type name-of-signature =
  sig
any sequence of type definitions
  end;;
```

The way you tell OCaML that this is not a definition of a module is by saying `module type`. This definition is abstract, just telling OCaML what kind of functions and objects reside in a module of this type. The statements that you may put there are `type` and `val`. The first declares a ground type (which can actually also be abstract!), and the second a derived type of a function. Both can be empty. If you look into `list.mli` you will find that it declares no types, only functions. To see a complex example of types and type definitions, take a look at `set.mli`.

OCaML is partitioned into three parts: the core language, the basic language, and extensions. Everything that does not belong to the core language is added in the form of a module. The manual lists several dozens of such modules. When you invoke OCaML it will automatically make available the core and everything in the basic language. Since you are always free to add more modules, the manual does not even attempt to list all modules, instead it lists the more frequent ones (that come with the distribution).

6 Sets and Functors

Perhaps the most interesting type constructor is that of sets. In type theory we can simply say that like for lists there is a unary constructor, s such that

(44) $$M_{s(\alpha)} = \wp(M_\alpha)$$

However, this is not the way it can be done in OCaML. The reason is that OCaML needs to be told how the elements of a set are ordered. For OCaML will internally store a set as a binary branching tree so that it can search for elements efficiently. To see why this is necessary, look at the way we write sets. We write sets linearly. This, however, means that we pay a price. We have to write, for example, $\{\varnothing, \{\varnothing\}\}$. The two occurrences of $\varnothing$ are two occurrences of the same set, but you have to

write it down twice since the set is written out linearly. In the same way, OCaML stores sets in a particular way, here in form of a binary branching tree. Next, OCaML demands from you that you order the elements linearly, in advance. You can order them in any way you please, but given two distinct elements a and b, either $a < b$ or $b < a$ must hold. This is needed to access the set, to define set union, and so on. The best way to think of a set as being a list of objects ordered in a strictly ascending sequence. If you want to access an element, you can say: take the least of the elements. This picks out an element. And it picks out exactly one. The latter is important because OCaML operates deterministically. Every operation you define must be total and deterministic.

If elements must always be ordered—how can we arrange the ordering? Here is how.

(45)
```
module OPStrings =
  struct
    type t = string * string
    let compare x y =
      if x = y then 0
      else if fst x > fst y || (fst x = fst y
            && snd x > snd y) then 1
      else -1
  end;;
```

(The indentation is just for aesthetic purposes and not necessary.) This is what OCaML answers:

(46)
```
module OPStrings :
sig type t = string * string val compare :
    'a * 'b -> 'a * 'b -> int end
```

It says that there is now a module called `OPString` with the following **signature**: there is a type `t` and a function `compare`, and their types inside the signature are given. A signature, by the way, is a set of functions together with their types. The signature is given inside `sig`$\cdots$`end`.

Now what is this program doing for us? It defines a module, which is a complete unit that has its own name. We have given it the name `OPStrings`. The

definition is given enclosed in `struct···end;;`. First, we say what entity we make the members from. Here they are strings. Second, we define the function `compare`. It must have integer values; its value is 0 if the two are considered equal, 1 if the left argument precedes the right hand argument and -1 otherwise. To define the predicate `compare` we make use of two things: first, strings can be compared; there is an ordering predefined on strings. Then we use the projection functions to access the first and the second component.

Finally, to make sets of the desired kind, we issue

(47) `module PStringSet = Set.Make(OPStrings);;`

To this, OCaML answers with a long list. This is because `Make` is a *functor*. A **functor** is a function that makes new modules from existing modules. They are thus more abstract than modules. The functor `Make`, defined inside the module `Set`, allows to create sets over entities of any type. It imports into `PStringSet` all set functions that it has predefined in the module `Set`, for example `Set.add`, which adds an element to a set or `Set.empty`, which denotes the empty set. However, the functions are now called `PStringSet`. For example, the empty set is called `PStringSet.empty`. Now, type

(48) `# st = PStringSet.empty;;`

and this assigns `st` to the empty set. If you want to have the set {(`cat`,`mouse`)} you can issue

(49) `# PStringSet.add ("cat", "mouse") st;;`

Alternatively, you can say

(50) `# PStringSet.add ("cat", "mouse") PStringSet.empty;;`

However, what will not work is if you type

(51) `# let st = PStringSet.add ("cat", "mouse") st;;`

This is because OCaML does not have dynamic assignments like imperative languages. `st` has been assigned already. You cannot change its binding. How dynamic assignments can be implemented will be explained later. We only mention the following. The type of sets is `PStringSet.t`. It is abstract. If you want

to actually see the set, you have to tell OCaML how to show it to you. One way of doing that is to convert the set into a list. There is a function called `elements` that converts the set into a list. Since OCaML has a predefined way of communicating sets (which we explained above), you can now look at the elements without trouble. However, what you are looking at are members of a list, not that of the set from which the list was compiled. This can be a source of mistakes in programming. Now if you type, say,

(52)
```
# let h = PStringSet.element st;;
```

OCaML will incidentally give you the list. It is important to realize that the program has no idea how it make itself understood to you if you ask it for the value of an object of a newly defined type. You have to tell it how you want it to show you.

Also, once sets are defined, a comparison predicate is available. That is to say, the sets are also ordered linearly by `PStringSet.compare`. This is useful, for it makes it easy to define sets of sets. Notice that `PStringSet.compare` takes its arguments to the right. The argument immediately to its right is the one that shows up to the right in infix notation. So, `PStringSet.compare f g` is the same as `g < f` in normal notation. Beware!

7 Hash Tables

This section explains some basics about hash tables. Suppose there is a function, which is based on a finite look up table (so, there are finitely many possible inputs) and you want to compute this function as fast as possible. This is the moment when you want to consider using hash tables. They are some implementation of a fact look up procedure. You make the hash table using magical incantations similar to those for sets. First you need to declare from what kind of objects to what kind of objects the function is working. Also, you sometimes (but not always) need to issue a function that assigns every input value a unique number (called **key**). So, you define first a module of inputs, after which you issue `HashTbl.make`. This looks as follows.

(53)
```
module HashedTrans =
  struct
```

```
    type t = int * char
    let equal x y = (((fst x = fst y) && (snd x = snd y)))
    let hash x = ((256 * (fst x)) + (int_of_char (snd x)))
  end;;

module HTrans = Hashtbl.Make(HashedTrans);;
```

8 Combinators

Combinators are functional expressions that use nothing but application of a function to another. They can be defined without any symbols except brackets. Functions can only be applied to one argument at a time. If f is a function and x some object, then fx or (fx) denotes the result of applying f to x. fx might be another function, which can be applied to something else. Then $(fx)y$ is the result of applying it to y. Brackets may be dropped in this case. Bracketing is left associative in general. This is actually the convention that is used in OCaML.

The most common combinators are I, K and S. They are defined as follows.

(54) $\mathsf{I}x := x$,

(55) $\mathsf{K}xy := x$,

(56) $\mathsf{S}xyz := xz(yz)$

This can be programmed straightforwardly as follows. (I give you a transcript of a session with comments by OCaML.)

(57)
```
# let i x = x;;
val i :  'a -> 'a = <fun>
# let k x y = x;;
val k :  'a -> 'b -> 'a = <fun>
# let s x y z = x z(y z)
val s :  ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
    = <fun>
```

Let us look carefully at what OCaML says. First, notice that it types every combinator right away, using the symbols `'a`, `'b` and so on. This is because as such they

can be used on any input. Moreover, OCaML returns the type using the right (!) associative bracketing convention. It drops as many brackets as it possibly can. Using maximal bracketing, the last type is

(58)
```
(('a -> ('b -> 'c)) -> (('a -> 'b) -> ('a -> 'c)))
```

Notice also that the type of the combinator is not entirely trivial. The arguments x, y and z must be of type `('a -> ('b -> 'c))`, `'a -> 'b` and `'c`, respectively. Otherwise, the right hand side is not well-defined. You may check, however, that with the type assignment given everything works well.

A few words on notation. Everything that you define and is not a concrete thing (like a string or a digit sequence) is written as a plain string, and the string is called an **identifier**. It identifies the object. Identifiers must always be unique. If you define an identifier twice, only the last assignment will be used. Keeping identifiers unique can be difficult, so there are ways to ensure that one does not make mistakes. There are certain conventions on identifiers. First, identifiers for functions are variables must be begin with a lower case letter. So, if you try to define combinators named `I`, `K` or `S`, OCaML will complain about syntax error. Second, if you have a sequence of identifiers, you must always type a space between them unless there is a symbol that counts as a separator (like brackets), otherwise the whole sequence will be read as a single identifier. Evidently, separators cannot be part of legal identifiers. For example, we typically write $\mathsf{S}xyz = xz(yz)$. But to communicate this to OCaML you must insert spaces between the variable identifiers except where brackets intervene. It is evident that an identifier cannot contain blanks nor brackets, otherwise the input is misread. (Read Page 89–90 on the issue of naming conventions. It incidentally tells you that comments are enclosed in `(*` and `*)`, with no intervening blanks between these two characters.) Finally, OCaML knows which is function and which is argument variable because the first identifier after `let` is interpreted as the value to be bound, and the sequence of identifiers up to the equation sign is interpreted as the argument variables.

Now, having introduced these combinators, we can apply them:

(59)
```
# k i 'a' 'b';;
- :  char = 'b'
```

This is because $\mathsf{Kia} = \mathsf{i}$ and $\mathsf{ib} = \mathsf{b}$. On the other hand, you may verify that

(60)
```
# k (i 'a') 'b';;
- :  char = 'a'
```

You can of course now define

(61)
```
# let p x y = k i x y;;
val p = 'a -> 'b -> 'b = <fun>
```

and this defines the function $pxy = y$. Notice that you cannot ask OCaML to evaluate a function abstractly. This is because it does not know that when you ask `k i x y` the letters `x` and `y` are variables. There are no free variables!

So, you can apply combinators to concrete values. You cannot calculate abstractly using the inbuilt functions of OCaML. One thing you can do, however, is check whether an expression is typable. For example, the combinator SII has no type. This is because no matter how we assign the primitive types, the function is not well–defined. OCaML has this to say:

(62)
```
# s i i;;
This expression has type ('a -> 'b') -> 'a -> 'b but
    is used here with type ('a -> 'b) -> 'a
```

How does it get there? It matches the definition of `s` and its first argument `i`. Since its first argument must be of type `'a -> 'b -> 'c`, this can only be the case if `'a = 'b -> 'c`. Applying the function yields an expression of type `('a -> 'b) -> 'a -> 'c`. Hence, the next argument must have type `('b -> 'c) -> 'b`. This is what OCaML expects to find. It communicates this using `'a` and `'b` in place of `'b` and `'c`, since these are variables and can be renamed at will. (Yes, OCaML does use variables, but these are variables over types, not over actual objects.) However, if this is the case, and the argument is actually `i`, then the argument actually has type `('b -> 'c) -> 'b -> 'c`. Once again, since OCaML did some renaming, it informs us that the argument has type `('a -> 'b) -> 'a -> 'b`. The types do not match, so OCaML rejects the input as not well formed.

Now, even though OCaML does not have free variables, it does have variables. We have met them already. Every `let`–clause defines some function, but this definition involves saying what it does on its arguments. We have defined `i`, `k`, and `s` in this way. The identifiers `x`, `y` and `z` that we have used there are unbound outside of the `let`–clause. Now notice that upon the definition of `k`, OCaML issued its type as `'a -> 'b -> 'a`. This means that it is a function from objects

of type 'a to objects of 'b -> 'a. It is legitimate to apply it to just one object:

(63)
```
# k "cat";;
- :  '_a -> string = <fun>
```

This means that if you give it another argument, the result is a string (because the result is "cat"). You can check this by assigning an identifier to the function and applying it:

(64)
```
# let katz = k "cat";;
val katz :  '_a -> string = <fun>
# katz "mouse";;
- :  string = "cat"
```

Effectively, this means that you define functions by abstraction; in fact, this is the way in which they are defined. However, the way in which you present the arguments may be important. (However, you can define the order of the arguments in any way you want. Once you have made a choice, you must strictly obey that order, though.)

9 Objects and Methods

The role of variables is played in OCaMl by **objects**. Objects are abstract data structures, they can be passed around and manipulated. However, it is important that OCaML has to be told every detail about an object, including how it is accessed, how it can communicate the identity of that object to you and how it can manipulate objects. Here is a program that defines a natural number:

(65)
```
class number =
  object
  val mutable x = 0
    method get = x
    method succ = x <- x + 1
    method assign d = x <- d
end;;
```

This does the following: it defines a data type `number`. The object of this type can be manipulated by three so–called **methods**: `get`, which gives you the value, `succ` which adds 1 to the value, and `assign` which allows you to assign any number you want to it. But it must be an integer; you may want to find out how OCaML knows that `x` must be integer. For this is what it will say:

(66)
```
class number :
  object
    val mutable x :  int
    method assign int -> <unit>
    method get :  int
    method succ :  <unit>
end
```

Notice that it uses the type `<unit>`. This is the type that has no object associated with it. You may think of it as the type of actions. For `succ` asks OCaML to add the number 1 to the number.

Once we have defined the class, we can have as many objects of that class as we want. Look at the following dialog.

(67)
```
# let m = new number;;
val m :  number = <obj>
# m#get;;
- :  int = 0
```

The first line binds the identifier `m` to an object of class `number`. OCaML repeats this. The third line asks for the value of `m`. Notice the syntax: the value of `m` is not simply `m` itself. The latter is an object (it can have several parameters in it). So, if you want to see the number you have just defined, you need to type `m#get`. The reason why this works is that we have defined the method `get` for that class. If you define another object, say `pi`, then its value is `pi#get`. And the result of this method is the number associated with `m`. Notice that the third line of the class definition (**??**) asserts that the object contains some object `x` which can be changed (it is declared mutable) and which is set to 0 upon the creation of the object.

If you define another object `q` and issue, for example `q#succ` and then ask for

the value you get 1:

(68)
```
# let q = new number;;
val q :  number = <obj>
# q#succ;;
- :  unit = ()
# q#get;;
- :  int = 1
```

Notice the following. If you have defined an object which is a set, issuing `method get = x` will not help you much in seeing what the current value is. You will have to say, for example, `method get = PStringSet.elements x` if the object is basically a set of pairs of strings as defined above.

10 Characters, Strings and Regular Expressions

We are now ready to do some actual theory and implementation of linguistics. We shall deal first with strings and then with finite state automata. Before we can talk about strings, some words are necessary on characters. Characters are drawn from a special set, which is also referred to as the **alphabet**. In principle the alphabet can be anything, but in actual implementations the alphabet is always fixed. OCaML, for example, is based on the character table of ISO Latin 1 (also referred to as ISO 8859-1). It is included on the web page for you to look at. You may use characters from this set only. In theory, any character can be used. However, there arises a problem of communication with OCaML. There are a number of characters that do not show up on the screen, like carriage return. Other characters are used as delimiters (such as the quotes). It is for this reason that one has to use certain naming conventions. They are given on Page 90 – 91 of the manual. If you write \ followed by 3 digits, this accesses the ASCII character named by that sequence. OCaML has a module called `Char` that has a few useful functions. The function `code`, for example, converts a character into its 3–digit code. Its inverse is the function `chr`. Type `Char.code 'L';;` and OCaML gives you 76. So, the string `\076` refers to the character `L`. You can try out that function and see that it actually does support the full character set of ISO Latin 1. Another issue is of course your editor: in order to put in that character, you have to learn how your editor lets you do this. (In vi, you type either ¡Ctrl¡V and then the numeric code

as defined in ISO Latin 1 (this did not work when I tried it) or ¡Ctrl¡K and then a two-keystroke code for the character (this did work for me). If you want to see what the options are, type `:digraphs` or simply `:dig` in commmand mode and you get the list of options.)

It is not easy to change the alphabet in dealing with computers. Until Unicode becomes standard, different alphabets must be communicated using certain combinations of ASCII–symbols. For example, HTML documents use the combination `&auml;` for the symbol ä. As far as the computer is concerned this is a sequence of characters. But some programs allow you to treat this as a single character. Practically, computer programs dealing with natural language read the input first through a so–called **tokenizer**. The tokenizer does nothing but determine which character sequence is to be treated as a single symbol. Or, if you wish, the tokenizer translates strings in the standard alphabet into strings of an arbitrary, user defined alphabet. The Xerox tools for finite state automata, for example, allow you to define any character sequence as a token. This means that if you have a token of the form `ab` the sequence `abc` could be read as a string of length three, `a` followed by `b` followed by `c`, or as a string of two tokens, `ab` followed by `c`. Careful thought has to go into the choice of token strings.

Now we start with strings. OCaML has a few inbuilt string functions, and a module called `String`, which contains a couple of useful functions to manipulate strings. Mathematically, a string of length n is defined as a function from the set of numbers 0, 1, 2, ..., $n-1$ into the alphabet. The string `"cat"`, for example, is the function f such that $f(0) =$ `c`, $f(1) =$ `a` and $f(2) =$ `t`. OCaML uses a similar convention.

(69)
```
# let u = "cat";;
val u :  string = "cat"
# u.[1];;
char = 'a'
```

So, notice that OCaML starts counting with 0, as we generally do in this course. The first letter in `u` is addressed as `u.[0]`, the second as `u.[1]`, and so on. Notice that technically $n = 0$ is admitted. This is a string that has no characters in it. It is called the **empty string**. Here is where the naming conventions become really useful. The empty string can be denoted by `""`. Without the quotes you would not see anything. And OCaML would not either. We define the following operations on strings:

Definition 2 *Let $\vec{u}$ be a string. The length of $\vec{u}$ is denoted by $|\vec{u}|$. Suppose that $|\vec{u}| = m$ and $|\vec{v}| = n$. Then $\vec{u}^\frown\vec{v}$ is a string of length $m + n$, which is defined as follows.*

$$(\vec{u}^\frown\vec{v})(j) = \begin{cases} \vec{u}(j) & \text{if } j < m, \\ \vec{v}(j-m) & \text{else.} \end{cases} \tag{70}$$

*$\vec{u}$ is a **prefix** of $\vec{v}$ if there is a $\vec{w}$ such that $\vec{v} = \vec{u}^\frown\vec{w}$, and a postfix if there is a $\vec{w}$ such that $\vec{v} = \vec{w}^\frown\vec{u}$. $\vec{u}$ is a **substring** if there are $\vec{w}$ and $\vec{x}$ such that $\vec{v} = \vec{w}^\frown\vec{v}^\frown\vec{x}$.*

OCaML has a function `String.length` that returns the length of a given string. For example, `String.length "cat"` will give 3. Notice that by our conventions, you cannot access the symbol with number 3. Look at the following dialog.

(71)
```
# "cat".[2];;
- :  char = 't'
# "cat".[String.length "cat"];;
Exception:  Invalid_argument "String.get".
```

The last symbol of the string has the number 2, but the string has length 3. If you try to access an element that does not exist, OCaML raises the exception `Invalid_argument "String.get"`.

In OCaML, `^` is an infix operator for string concatenation. So, if we write `"tom"^"cat"` OCaML returns `"tomcat"`. Here is a useful abbreviation. $\vec{x}^n$ denotes the string obtained by repeating $\vec{x}$ n–times. This can be defined recursively as follows.

$$\vec{x}^0 = \varepsilon \tag{72}$$

$$\vec{x}^{n+1} = \vec{x}^n{}^\frown\vec{x} \tag{73}$$

So, $\mathtt{vux}^3 = \mathtt{vuxvuxvux}$.

A **language** over A is a set of strings over A. Here are a few useful operations on languages.

$$L \cdot M := \{\vec{x}\vec{y} : \vec{x} \in L, \vec{y} \in M\} \tag{74a}$$

$$L/M := \{\vec{x} : \text{exists } \vec{y} \in M\text{: } \vec{x}\vec{y} \in L\} \tag{74b}$$

$$M\backslash L := \{\vec{x} : \text{exists } \vec{y} \in M\text{: } \vec{y}\vec{x} \in L\} \tag{74c}$$

The first defines the language that contains all concatenations of elements from the first, and elements from the second. This construction is also used in 'switchboard' constructions. For example, define

(75) $L := \{\texttt{John}, \texttt{my neighbour}, \dots\}$

the set of noun phrases, and

(76) $M := \{\texttt{sings}, \texttt{rescued the queen}, \dots\}$

then $L \cdot \{\square\} \cdot M$ ($\square$ denotes the blank here) is the following set:

(77) $\{\texttt{John sings}, \texttt{John rescued the queen},$
$\texttt{my neighbour sings}, \texttt{my neighbour rescued the queen}, \dots\}$

Notice that (**??**) is actually not the same as $L \cdot M$. Secondly, notice that there is no period at the end, and no uppercase for `my`. Thus, although we do get what looks like sentences, some orthographic conventions are not respected.

The construction L/M denotes the set of strings that will be L–strings if some M–string is appended. For example, let $L = \{\texttt{chairs}, \texttt{cars}, \texttt{cats}, \dots\}$ and $M = \{\texttt{s}\}$. Then

(78) $L/M = \{\texttt{chair}, \texttt{car}, \texttt{cat}, \dots\}$

Notice that if L contains a word that does not end in `s`, then no suffix from M exists. Hence

(79) $\{\texttt{milk}, \texttt{papers}\}/M = \{\texttt{paper}\}$

In the sequel we shall study first regular languages and then context free languages. Every regular language is context free, but there are also languages that are neither regular nor context free. These languages will not be studied here.

A regular expression is built from letters using the following additional symbols: 0, ε, $\cdot$, $\cup$, and *. 0 and ε are constants, as are the letters of the alphabet. $\cdot$ and $\cup$ are binary operations, * is unary. The language that they specify is given as follows:

(80) $L(0) := \varnothing$

(81) $L(\varnothing) := \{\varnothing\}$

(82) $L(s \cdot t) := L(s) \cdot L(t)$

(83) $L(s \cup t) := L(s) \cup L(t)$

(84) $L(s^*) := \{x^n : x \in L(s), n \in \mathbb{N}\}$

($\mathbb{N}$ is the set of natural numbers. It contains all integers starting from 0.)$\cdot$ is often omitted. Hence, st is the same as $s \cdot t$. Hence, $\mathtt{cat} = \mathtt{c} \cdot \mathtt{a} \cdot \mathtt{t}$. It is easily seen that every set containing exactly one string is a regular language. Hence, every set containing a finite set of strings is also regular. With more effort one can show that a set containing all but a finite set of strings is regular, too.

Notice that whereas s is a term, $L(s)$ is a language, the language of terms that fall under the term s. In ordinary usage, these two are not distinguished, though. We write $\mathtt{a}$ both for the string $\mathtt{a}$ and the regular term whose language is $\{\mathtt{a}\}$. It follows that

(85) $L(\mathtt{a} \cdot (\mathtt{b} \cup \mathtt{c})) = \{\mathtt{ab}, \mathtt{ac}\}$

(86) $L((\mathtt{cb})^*\mathtt{a}) = \{\mathtt{a}, \mathtt{cba}, \mathtt{cbcba}, \dots\}$

A couple of abbreviations are also used:

(87) $s? := \varepsilon \cup s$

(88) $s^+ := s \cdot s^*$

(89) $s^n := s \cdot s \cdot \dots \cdot s$ (n–times)

We say that $s \subseteq t$ if $L(s) \subseteq L(t)$. This is the same as saying that $L(s \cup t) = L(t)$. Regular expressions are used in a lot of applications. For example, if you are searching for a particular string, say `department` in a long document, it may actually appear in two shapes, `department` or `Department`. Also, if you are searching for two words in a sequence, you may face the fact that they appear on different lines. This means that they are separated by any number of blanks and an optional carriage return. In order not to loose any occurrence of that sort you will want to write a regular expression that matches any of these occurrences. The Unix command `egrep` allows you to search for strings that match a regular term. The particular constructors look a little bit different, but the underlying concepts are the same.

Notice that there are regular expressions which are different but denote the same language. We have in general the following laws. (A note of caution. These laws are not identities between the terms; the terms are distinct. These are identities concerning the languages that these terms denote.)

(90a) $s \cdot (t \cdot u) = (s \cdot t) \cdot u$

(90b) $s \cdot \varepsilon = s$ $\varepsilon \cdot s = s$

(90c) $s \cdot 0 = 0$ $\qquad$ $0 \cdot s = 0$

(90d) $s \cup (t \cup u) = (s \cup t) \cup u$

(90e) $s \cup t = t \cup s$

(90f) $s \cup s = s$

(90g) $s \cup 0 = s$ $\qquad$ $0 \cup s = s$

(90h) $s \cdot (t \cup t) = (s \cdot t) \cup (s \cdot u)$

(90i) $(s \cup t) \cdot u = (s \cdot u) \cup (t \cdot u)$

(90j) $s^* = \varepsilon \cup s \cdot s^*$

It is important to note that the regular terms appear as solutions of very simple equations. For example, let the following equation be given. ($\cdot$ binds stronger than $\cup$.)

(91) $X = \mathsf{a} \cup \mathsf{b} \cdot X$

Here, the variable X denotes a language, that is, a set of strings. We ask: what set X satisfies the equation above? It is a set X such that a is contained in it and if a string $\vec{v}$ is in X, so is $\mathsf{b}^\frown\vec{v}$. So, since it contains a, it contains ba, bba, bbba, and so on. Hence, as you may easily verify,

(92) $X = \mathsf{b}^*\mathsf{a}$

is the smallest solution to the equation. (There are many more solutions, for example $\mathsf{b}^*\mathsf{a} \cup \mathsf{b}^*$. The one given is the smallest, however.)

Theorem 3 *Let s and t be regular terms. Then the smallest solution of the equation $X = t \cup s \cdot X$ is $s^* \cdot t$.*

Now, $s \cdot t$ is the unique solution of $Y = t, X = s \cdot t$; $s \cup t$ is the solution of $X = Y \cup Z$, $Y = s$, $Z = t$. Using this, one can characterize regular languages as minimal solutions of certain systems of equations.

Let X_i, $i < n$, be variables. A regular equation has the form

(93) $X_i = s_0 \cdot X_0 \cup s_1 \cdot X_1 \ldots \cup s_{n-1} \cdot X_{n-1}$

(If s_i is 0, the term $s_i \cdot X_i$ may be dropped.) The equation is **simple** if for all $i < n$, s_i is either 0, ε, or a single letter. The procedure we have just outlined can be used to convert any regular expression into a set of simple equations whose minimal solution is that term. The converse is actually also true.

Theorem 4 *Let E be a set of regular equations in the variables X_i, $i < n$, such that X_i is exactly once to the left of an equation. Then there are regular terms s_i, $i < n$, such that the least solution to E is $X_i = L(s_i)$, $i < n$.*

Notice that the theorem does not require the equations of the set to be simple. The proof is by induction on n. We suppose that it is true for a set of equations in $n-1$ letters. Suppose now the first equation has the form

$$X_0 = s_0 \cdot X_0 \cup s_1 \cdot X_1 \ldots \cup s_{n-1} \cdot X_{n-1} \tag{94}$$

Two cases arise. Case (1). $s_0 = 0$. Then the equation basically tells us what X_0 is in terms of the X_i, $0 < i < n$. We solve the set of remaining equations by hypothesis, and we obtain regular solutions t_i for X_i. Then we insert them:

$$X_0 = s_1 \cdot t_1 \cup s_1 \cdot t_1 \cup \ldots \cup s_{n-1} \cdot t_{n-1} \tag{95}$$

This is a regular term for X_0. Case (2). $s_0 \neq 0$. Then using Theorem **??** we can solve the equation by

$$\begin{aligned} X_0 &= s_0^*(s_1 \cdot X_1 \ldots \cup s_{n-1} \cdot X_{n-1}) \\ &= s_0^* s_1 \cdot X_1 \cup s_0^* s_2 \cdot X_2 \cup \ldots \cup s_0^* s_{n-1} \cdot X_{n-1} \end{aligned} \tag{96}$$

and proceed as in Case (1).

The procedure is best explained by an example. Take a look at Table **??**. It shows a set of three equations, in the variables X_0, X_1 and X_2. (This is (I).) We want to find out what the minimal solution is. First, we solve the second equation for X_1 with the help of Theorem **??** and get (II). The simplification is that X_1 is now defined in terms of X_0 alone. The recursion has been eliminated. Next we insert the solution we have for X_1 into the equation for X_2. This gives (III). Next, we can also put into the first equation the solution for X_2. This is (IV). Now we have to solve the equation for X_0. Using Theorem **??** again we get (V). Now X_0 given explicitly. This solution is finally inserted in the equations for X_1 and X_2 to yield (VI).

11 Interlude: Regular Expressions in OCaML

OCaML has a module called `Str` that allows to handle regular expressions, see Page 397. If you have started OCaML via the command `ocaml` you must make

Table 1: Solving a set of equations

$$
\begin{array}{lllll}
\text{(I)} & X_0 = & \varepsilon & & \cup \mathtt{d} X_2 \\
& X_1 = & \mathtt{b} X_0 & \cup \mathtt{d} X_1 & \\
& X_2 = & & \mathtt{c} X_1 &
\end{array}
$$

$$
\begin{array}{lllll}
\text{(II)} & X_0 = & \varepsilon & & \cup \mathtt{d} X_2 \\
& X_1 = & \mathtt{d}^* \mathtt{b} X_0 & & \\
& X_2 = & & \mathtt{c} X_1 &
\end{array}
$$

$$
\begin{array}{lllll}
\text{(III)} & X_0 = & \varepsilon & & \cup \mathtt{d} X_2 \\
& X_1 = & \mathtt{d}^* \mathtt{b} X_0 & & \\
& X_2 = & \mathtt{c}\mathtt{d}^* \mathtt{b} X_0 & &
\end{array}
$$

$$
\begin{array}{lll}
\text{(IV)} & X_0 = & \varepsilon \quad \cup \mathtt{d}\mathtt{c}\mathtt{d}^* \mathtt{b} X_0 \\
& X_1 = & \mathtt{d}^* \mathtt{b} X_0 \\
& X_2 = & \mathtt{c}\mathtt{d}^* \mathtt{b} X_0
\end{array}
$$

$$
\begin{array}{lll}
\text{(V)} & X_0 = & \mathtt{d}\mathtt{c}\mathtt{d}^* \mathtt{b} \\
& X_1 = & \mathtt{d}^* \mathtt{b} X_0 \\
& X_2 = & \mathtt{c}\mathtt{d}^* \mathtt{b} X_0
\end{array}
$$

$$
\begin{array}{lll}
\text{(VI)} & X_0 = & \mathtt{d}\mathtt{c}\mathtt{d}^* \mathtt{b} \\
& X_1 = & \mathtt{d}^* \mathtt{b}\mathtt{d}\mathtt{c}\mathtt{d}^* \mathtt{b} \\
& X_2 = & \mathtt{c}\mathtt{d}^* \mathtt{b}\mathtt{d}\mathtt{c}\mathtt{d}^* \mathtt{b}
\end{array}
$$

sure to bind the module into the program. This can be done interactively by

(97) `# #load "str.cma";;`

The module provides a type `regexp` and various functions that go between strings and regular expressions. The module actually ”outsources” the matching and uses the matcher provided by the platform (POSIX in Unix). This has two immediate effects: first, different platforms implement different matching algorithms and this may result in different results of the same program; second, the syntax of the regular expressions depends on the syntax of regular expressions of the matcher that the platform provides. Thus, if you enter a regular expression, there is a two stage conversion process. The first is that of the string that you give to OCaML into a regular term. The conventions are described in the manual. The second is the conversion from the regular term of OCaML into one that the platform uses. This is why the manual does not tell you exactly what the exact syntax of regular expressions is. In what is to follow, I shall describe the syntax of regular expressions in Unix (they are based on Gnu Emacs).

A regular expression is issued to OCaML as a string. However, the string itself is not seen by OCaML as a regular expression; to get the regular expression you have to apply the function `regexp` (to do that, you must, of course, type `Str.regexp`). It is worthwhile to look at the way regular expressions are defined. First, the characters `$^.*+?\[]` are special and set aside. Every other symbol is treated as a constant for the character that it is. (We have used the same symbolic overload above. The symbol `a` denotes not only the character `a` but also the regular expression whose associated language is {`a`}.) Now here are a few of the definitions.

- `.` matches any character except newline
- `*` (postfix) translates into *.
- `+` (postfix) translates into $^+$.
- `?` (postfix) translates into ?.
- `^` matches at the beginning of the line.
- `$` matches at the end of line.
- `\\(` left (opening) bracket.

- `\\)` right (closing) bracket.
- `\\|` translates into ∪.

Thus, (a ∪ b)$^+$d? translates into `\\(a\\|b\\)+d?`. Notice that in OCaML there is also a way to quote a character by using \ followed by the 3–digit code, and this method can also be used in regular expressions. For example, the code of the symbol + is 43. Thus, if you are looking for a nonempty sequence of +, you have to give OCaML the following string: `\043+`. Notice that ordinarily, that is, in a string, the sequence `\043` is treated in the same way as +. Here, however, we get a difference. The sequence `\043` is treated as a character, while + is now treated as an operation symbol. It is clear that this syntax allows us to define any regular language over the entire set of characters, including the ones we have set aside.

There are several shortcuts. One big group concerns grouping together characters. To do this, one can use square brackets. The square brackets eliminate the need to use the disjunction sign. Also it gives additional power, as we shall explain now. The expression `[abx09]` denotes a single character matching any of the symbols occuring in the bracket. Additionally, we can use the hyphen to denote from-to: `[0-9]` denotes the digits, `[a-f]` the lower case letters from 'a' to 'f', `[A-F]` the upper case letters from 'A' to 'F'; finally, `[a-fA-F]` denotes the lower and upper case letters from 'a' to 'f' ('A' to 'F'). If the opening bracket is immediately followed by a caret, the range is inverted: we now take anything that is *not* contained in the list. Notice that ordinarily, the hyphen is a string literal, but when it occurs inside the square brackets it takes a new meaning. If we do want the hyphen to be included in the character range, it has to be put at the beginning or the end. If we want the caret to be a literal it must be placed last.

Now that we know how to write regular expressions, let us turn to the question how they can be used. By far the most frequent application of matching against a regular expression is not that of an exact match. Rather, one tries to find an occurrence of a string that falls under the regular expression in a large file. For example, if you are editing a file and you what to look to the word `ownership` in a case insensitive way, you will have to produce a regular expression that matches either `ownership` or `Ownership`. Similarly, looking for a word in German that may have ä or `ae`, ü next to `ue` and so on as alternate spellings may prompt the need to use regular expressions. The functions that OCaML provides are geared towards this application. For example, the function `search_forward` needs a regular expression as input, then a string and next an integer. The integer specifies

the position in the string where the search should start. The result is an integer and it gives the initial position of the next string that falls under the regular expression. If none can be found, OCaML raises the exception `Not_found`. If you want to avoid getting the exception you can also use `string_match` before to see whether or not there is any string of that sort. It is actually superfluous to first check to see whether a match exists and then actually performing the search. This requires to search the same string twice. Rather, there are two useful functions, `match_beginning` and `match_end`, which return the initial position (final position) of the string that was found to match the regular expression. Notice that you have to say `Str.match_beginning()`, supplying an empty parenthesis.

There is an interesting difference in the way matching can be done. The Unix matcher tries to match the longest possible substring against the regular expression (this is called **greedy matching**), while the Windows matcher looks for the just enough of the string to get a match (**lazy matching**). Say we are looking for $\texttt{a}^*$ in the string `bcagaa`.

(98)
```
let ast = Str.regexp "a*";;
Str.string_match ast "bcagaa" 0;;
```

In both versions, the answer is ”yes”. Moreover, if we ask for the first and the last position, both version will give 0 and 0. This is because there is no way to match more than zero `a` at that point. Now try instead

(99)
```
Str.string_match ast "bcagaa" 2;;
```

Here, the two algorithms yield different values. The greedy algorithm will say that it has found a match between 2 and 3, since it was able to match one successive `a` at that point. The lazy algorithm is content with the empty string. The empty string matches, so it does not look further. Notice that the greedy algorithm does not look for the longest occurrence in the entire string. Rather, it proceeds from the starting position and tries to match from the successive positions against the regular expression. If it cannot match at all, it proceeds to the next position. If it can match, however, it will take the longest possible string that matches.

The greedy algorithm actually allows to check whether a string falls under a

regular expression. The way to do this is as follows.

```
(100)  let exact_match r s =
           if Str.string_match r s 0
           then ((Str.match_beginning () = 0)
               && (Str.match_end () = (String.length s)))
           else false;;
```

The most popular application for which regular expressions are used are actually string replacements. String replacements work in two stages. The first is the phase of matching. Given a regular expression s and a string $\vec{x}$, the pattern matcher looks for a string that matches the expression s. It always looks for the first possible match. Let this string be $\vec{y}$. The next phase is the actual replacement. This can be a simple replacement by a particular string. Very often we do want to use parts of the string $\vec{y}$ in the new expression. An example may help.

An IP address is a sequence of the form $\vec{c}.\vec{c}.\vec{c}.\vec{c}$, where $\vec{c}$ is a string representing a number from 0 to 255. Leading zeros are suppressed, but $\vec{c}$ may not be empty. Thus, we may have `192.168.0.1`, `145.146.29.243`, of `127.0.0.1`. To define the legal sequences, define the following string:

```
(101)  [01][0-9][0-9]| 2[0-4][0-9]| 25[0-4]
```

This string can be converted to a regular expression using `Str.regexp`. It matches exactly the sequences just discussed. Now look at the following:

```
(102)  let x = "[01][0-9][0-9] \\| 2[0-4][0-9]\\| 25[0-4]"
       let m = Str.regexp "\\("^x^"\\."^"\\)"^x^"\\.
           "^"\\("^x^"\\."^"\\)"
```

This expression matches IP addresses. The bracketing that I have introduced can be used for the replacement as follows. The matcher that is sent to find a string that matches a regular expression actually takes note for each bracketed expression where it matched in the string that it is looking at. It will have a record that says, for example, that the first bracket matched from position 1230 to position 1235, and the second bracket from position 1237 to position 1244. For example, if your data is as follows:

```
(103)  ...129.23.145.110...
```

Suppose that the first character is at position 1230. Then the string `129.023` matches the first bracket and the string `145.110` the second bracket. These strings can be recalled using the function `matched_group`. It takes as input a number and the original string, and it returns the string of the *n*th matching bracket. So, if directly after the match on the string assigned to `u` we define

(104)
```
let s = "The first half of the IP address is
    "^(Str.matched_group 1 u)
```

we get the following value for `s`:

(105)
```
"The first half of the IP address is 129.23
```

To use this in an automated string replacement procedure, the variables `\\0`, `\\1`, `\\2`,..., `\\9`. After a successful match, `\\0` is assigned to the entire string, `\\1`, to the first matched string, `\\2` to the second matched string, and so on. A **template** is a string that in place of characters also contains these variables (but nothing more). The function `global_replace` takes as input a regular expression, and two strings. The first string is used as a template. Whenever a match is found it uses the template to execute the replacement. For example, to cut the IP to its first half, we write the template `"\\1"`. If we want to replace the original IP address by its first part followed by .0.1, then we use `"\\.0.1"`. If we want to replace the second part by the first, we use `"\\1.\\"`.

12 Finite State Automata

A **finite state automaton** is a quintuple

(106) $\mathfrak{A} = \langle A, Q, i_0, F, \delta \rangle$

where A, the **alphabet**, is a finite set, Q, the set of **states**, also is a finite set, $i_0 \in Q$ is the **initial state**, $F \subseteq Q$ is the set of **final** or **accepting states** and, finally, $\delta \subseteq Q \times A \times Q$ is the **transition relation**. We write $x \xrightarrow{a} y$ if $\langle x, a, y \rangle \in \delta$.

We extend the notation to regular expressions as follows.

(107a) $x \overset{0}{\longrightarrow} y :\Leftrightarrow$ false

(107b) $x \overset{\varepsilon}{\longrightarrow} y :\Leftrightarrow x = y$

(107c) $x \overset{s\cup t}{\longrightarrow} y :\Leftrightarrow x \overset{s}{\longrightarrow} y$ or $x \overset{t}{\longrightarrow} y$

(107d) $x \overset{st}{\longrightarrow} y :\Leftrightarrow$ exists z: $x \overset{s}{\longrightarrow} z$ and $z \overset{t}{\longrightarrow} y$

(107e) $x \overset{s^*}{\longrightarrow} y :\Leftrightarrow$ exists n: $x \overset{s^n}{\longrightarrow} y$

In this way, we can say that

(108) $L(\mathfrak{A}) := \{\vec{x} \in A^* : \text{there is } q \in F\text{: } i_0 \overset{\vec{x}}{\longrightarrow} q\}$

and call this the **language accepted by** $\mathfrak{A}$. Now, an automaton is **partial** if for some q and a there is no q' such that $q \overset{a}{\to} q'$.

Here is part of the listing of the file fstate.ml.

(109)
```
class automaton =
  object
    val mutable i = 0
    val mutable x = StateSet.empty
    val mutable y = StateSet.empty
    val mutable z = CharSet.empty
    val mutable t = Transitions.empty
    method get_initial = i
    method get_states = x
    method get_astates = y
    method get_alph = z
    method get_transitions = t
    method initialize_alph = z <- CharSet.empty
...
    method list_transitions = Transitions.elements t
  end;;
```

There is a lot of repetition involved in this definition, so it is necessary only to look at part of this. First, notice that prior to this definition, the program contains definitions of sets of states, which are simply sets of integers. The type is called `StateSet`. Similarly, the type `CharSet` is defined. Also there is the type of transition, which has three entries, `first`, `symbol` and `second`. They define the state prior to scanning the symbol, the symbol, and the state after scanning the symbol, respectively. Then a type of sets of transitions is defined. These are now used in the definition of the object type `automaton`. These things need to be formally introduced. The attribute `mutable` shows that their values can be changed. After the equation sign is a value that is set by default. You do not have to set the value, but to prevent error it is wise to do so. You need a method to even get at the value of the objects involved, and a method to change or assign a value to them. The idea behind an object type is that the objects can be changed, and you can change them interactively. For example, after you have loaded the file fsa.ml typing `#use "fsa.ml"` you may issue

(110) `# let a = new automaton;;`

Then you have created an object identified by the letter `a`, which is a finite state automaton. The initial state is 0, the state set, the alphabet, the set of accepting states is empty, and so is the set of transitions. You can change any of the values using the appropriate method. For example, type

(111) `# a#add_alph 'z';;`

and you have added the letter 'z' to the alphabet. Similarly with all the other components. By way of illustration let me show you how to add a transition.

(112) `# a#add_transitions {first = 0; second = 2; symbol = 'z'};;`

So, effectively you can define and modify the object step by step.

If $\mathfrak{A}$ is not partial it is called **total**. It is an easy matter to make an automaton total. Just add a state $q_\sharp$ and add a transition $\langle q, a, q_\sharp\rangle$ when there was no transition $\langle q, a, q'\rangle$ for any q'. Finally, add all transitions $\langle q_\sharp, a, q_\sharp\rangle$, $q_\sharp$ is not accepting.

Theorem 5 *Let $\mathfrak{A}$ be a finite state automaton. Then $L(\mathfrak{A})$ is regular.*

The idea is to convert the automaton into an equation. To this end, for every state q let $T_q := \{x : i_0 \stackrel{x}{\longrightarrow} q\}$. Then if $\langle r, a, q\rangle$, we have $T_q \supseteq a \cdot T_r$. In fact, if $i \neq i_0$ we

have

$$T_q = \bigcup_{\langle r,a,q\rangle\in\delta} T_r \cdot a \tag{113}$$

If $q = i_0$ we have

$$T_{i_0} = \varepsilon \cup \bigcup_{\langle r,a,q\rangle\in\delta} T_r \cdot a \tag{114}$$

This is a set of regular equations, and it has a solution t_i for every q, so that $T_q = L(t_q)$. Now,

$$L(\mathfrak{A}) = \bigcup_{q\in F} T_q \tag{115}$$

which is regular.

On the other hand, for every regular term there is an automaton $\mathfrak{A}$ that accepts exactly the language of that term. We shall give an explicit construction of such an automaton.

First, $s = 0$. Take an automaton with a single state, and put $F = \varnothing$. No accepting state, so no string is accepted. Next, $s = \varepsilon$. Take two states, 0 and 1. Let 0 be initial and accepting, and $\delta = \{\langle 0, a, 1\rangle, \langle 1, a, 1\rangle : a \in A\}$. For $s = a$, let $Q = \{0, 1, 2\}$, 0 initial, 1 accepting. $\delta = \{\langle 0, a, 1\rangle\} \cup \{\langle 1, b, 1\rangle : b \in A\} \cup \{\langle 0, b, 2\rangle : b \in A - \{a\}\} \cup \{\langle 2, b, 2\rangle : b \in A\}$. Next st. Take an automaton $\mathfrak{A}$ accepting s, and an automaton $\mathfrak{B}$ accepting t. The new states are the states of $\mathfrak{A}$ and the states $\mathfrak{B}$ (considered distinct) with the initial state of $\mathfrak{B}$ removed. For every transition $\langle i_0, a, j\rangle$ of $\mathfrak{B}$, remove that transition, and add $\langle q, a, j\rangle$ for every accepting state of $\mathfrak{A}$. The accepting states are the accepting states of $\mathfrak{B}$. (If the initial state was accepting, throw in also the accepting states of $\mathfrak{A}$.) Now for s^*. Make $F \cup \{i_0\}$ accepting. Add a transition $\langle u, a, j\rangle$ for every $\langle i_0, a, j\rangle \in \delta$ such that $u \in F$. This automaton recognizes s^*. Finally, $s \cup t$. We construct the following automaton.

$$\mathfrak{A} \times \mathfrak{B} = \langle A, Q^{\mathfrak{A}} \times Q^{\mathfrak{B}}, \langle i_0^{\mathfrak{A}}, i_0^{\mathfrak{B}}\rangle, F^{\mathfrak{A}} \times F^{\mathfrak{B}}, \delta^{\mathfrak{A}} \times \delta^{\mathfrak{B}}\rangle \tag{116}$$

$$\delta^{\mathfrak{A}} \times \delta^{\mathfrak{B}} = \{\langle x_0, x_1\rangle \xrightarrow{a} \langle y_0, y_1\rangle : \langle x_0, a, y_0\rangle \in \delta^{\mathfrak{A}}, \langle x_1, a, y_1\rangle \in \delta^{\mathfrak{B}}\} \tag{117}$$

Lemma 6 *For every string* $\vec{u}$

$$\langle x_0, x_1\rangle \xrightarrow{\vec{u}}_{\mathfrak{A}\times\mathfrak{B}} \langle y_0, y_1\rangle \quad \Leftrightarrow \quad x_0 \xrightarrow{\vec{u}}_{\mathfrak{A}} y_0 \text{ and } y_0 \xrightarrow{\vec{u}}_{\mathfrak{B}} y_1 \tag{118}$$

The proof is by induction on the length of $\vec{u}$. It is now easy to see that $L(\mathfrak{A} \times \mathfrak{B}) = L(\mathfrak{A}) \cap L(\mathfrak{B})$. For $\vec{u} \in L(\mathfrak{A} \times \mathfrak{B})$ if and only if $\langle i_0^{\mathfrak{A}}, i_0^{\mathfrak{B}} \rangle \stackrel{\vec{u}}{\to} \langle x_0, x_1 \rangle$, which is equivalent to $i_0^{\mathfrak{A}} \stackrel{\vec{u}}{\to} x_0$ and $i_0^{\mathfrak{B}} \stackrel{\vec{u}}{\to} x_1$. The latter is nothing but $\vec{u} \in L(\mathfrak{A})$ and $\vec{u} \in L(\mathfrak{B})$.

Now, assume that $\mathfrak{A}$ and $\mathfrak{B}$ are total. Define the set $G := F^{\mathfrak{A}} \times Q^{\mathfrak{B}} \cup Q^{\mathfrak{A}} \times F^{\mathfrak{B}}$. Make G the accepting set. Then the language accepted by this automaton is $L(\mathfrak{A}) \cup L(\mathfrak{B}) = s \cup t$. For suppose u is such that $\langle i_0^{\mathfrak{A}}, i_0^{\mathfrak{B}} \rangle \stackrel{\vec{u}}{\to} q \in G$. Then either $q = \langle q_0, y \rangle$ with $q_0 \in F^{\mathfrak{A}}$ and then $\vec{u} \in L(\mathfrak{A})$, or $q = \langle x, q_1 \rangle$ with $q_1 \in F^{\mathfrak{B}}$. Then $\vec{u} \in L(\mathfrak{B})$. The converse is also easy.

Here is another method.

Definition 7 *Let $L \subseteq A^*$ be a language. Then put*

$$L^p = \{\vec{x} : \textit{there is } \vec{y} \textit{ such that } \vec{x}^\frown \vec{y} \in L\} \tag{119}$$

Let s be a regular term. We define the prefix closure as follows.

$$0^\dagger := \varnothing \tag{120a}$$
$$a^\dagger := \{\varepsilon, a\} \tag{120b}$$
$$\varepsilon^\dagger := \{\varepsilon\} \tag{120c}$$
$$(s \cup t)^\dagger := s^\dagger \cup t^\dagger \tag{120d}$$
$$(st)^\dagger := \{su : u \in t^\dagger\} \cup s^\dagger \tag{120e}$$
$$(s^*)^\dagger := s^\dagger \cup \{s^* u : u \in s^\dagger\} \tag{120f}$$

Notice the following.

Lemma 8 *$\vec{x}$ is a prefix of some string from s iff $\vec{x} \in L(t)$ for some $t \in s^\dagger$. Hence,*

$$L(s)^p = \bigcup_{t \in s^\dagger} L(t) \tag{121}$$

Proof. Suppose that $t \in s^\dagger$. We show that $L(t)$ is the union of all $L(u)$ where $u \in s^\dagger$. The proof is by induction on s. The case $s = \varnothing$ and $s = \varepsilon$ are actually easy. Next, let $s = a$ where $a \in A$. Then $t = a$ or $t = \varepsilon$ and the claim is verified directly.

Next, assume that $s = s_1 \cup s_2$ and let $t \in s^\dagger$. Either $t \in s_1^\dagger$ or $t \in s_2^\dagger$. In the first case, $L(t) \subseteq L(s_1)^p$ by inductive hypothesis, and so $L(t) \subseteq L(s)^p$, since $L(s) \supseteq L(s_1)$ and so $L(s)^p \supseteq L(s_1)^p$. Analogously for the second case. Now, if $\vec{x} \in L(s)^p$ then either $\vec{x} \in L(s_1)^p$ or $L(s_2)^p$ and hence by inductive hypothesis $\vec{x} \in u$ for some $u \in s_1^\dagger$ or some $u \in s_2^\dagger$, so $u \in s^\dagger$, as had to be shown.

Next let $s = s_1 \cdot s_2$ and $t \in s^\dagger$. Case 1. $t \in s_1^\dagger$. Then by inductive hypothesis, $L(t) \subseteq L(s_1)^p$, and since $L(s_1)^p \subseteq L(s)^p$ (can yous ee this?), we get the desired conclusion. Case 2. $t = s \cdot u$ with $u \in L(s_2)^\dagger$. Now, a string in $L(t)$ is of the form $\vec{x}\vec{y}$, where $\vec{x} \in L(s_1)$ and $\vec{y} \in L(u) \subseteq L(s_2)^p$, by induction hypothesis. So, $\vec{x} \in L(s_1)L(s_2)^p \subseteq L(s)^p$, as had to be shown. Conversely, if a string is in $L(s)^p$ then either it is of the form $\vec{x} \in L(s_1)^p$ or $\vec{y}\vec{z}$, with $\vec{y} \in L(s_1)$ and $\vec{z} \in L(s_2)^p$. In the first case there is a $u \in s_1^\dagger$ such that $\vec{x} \in L(u)$; in the second case there is a $u \in s_2^\dagger$ such that $\vec{z} \in L(u)$ and $\vec{y} \in L(s_1)$. In the first case $u \in s^\dagger$; in the second case, $\vec{x} \in L(s_1 u)$ and $s_1 u \in s^\dagger$. This shows the claim.

Finally, let $s = s_1^*$. Then either $t \in s_1^\dagger$ or $t = s_1^* u$ where $u \in s_1^\dagger$. In the first case we have $L(t) \subseteq L(s_1)^p \subseteq L(s)^p$, by inductive hypothesis. In the second case we have $L(t) \subseteq L(s_1^*)L(u) \subseteq L(s_1^*)L(s_1)^p \subseteq L(s)^p$. This finishes one direction. Now, suppose that $\vec{x} \in L(s)$. Then there are $\vec{y}_i$, $i < n$, and $\vec{z}$ such that $\vec{y}_i \in L(s_1)$ for all $i < n$, and $\vec{z} \in L(s_1)^p$. By inductive hypothesis there is a $u \in L(s_1)$ such that $\vec{z} \in L(u)$. Also $\vec{y}_0\vec{y}_1 \cdots \vec{y}_{n-1} \in L(s_1^*)$, so $\vec{x} \in L(s_1^* u)$, and the regular term is in $s^\dagger$. □

We remark here that for all $s \neq \emptyset$: $\varepsilon \in s^\dagger$; moreover, $s \in s^\dagger$. Both are easily established by induction.

Let s be a term. For $t, u \in s^\dagger$ put $t \xrightarrow{a} u$ iff $ta \subseteq u$. (The latter means $L(ta) \subseteq L(u)$.) The start symbol is ε.

$$\begin{aligned} A(s) &:= \{a \in A : a \in s^\dagger\} \\ \delta(t, a) &:= \{u : ta \subseteq u\} \\ \mathfrak{A}(s) &:= \langle A(s), s^\dagger, \varepsilon, \delta\rangle \end{aligned} \tag{122}$$

canonical automaton of s. We shall show that the language recognized by the canonical automaton is s. This follows immediately from the next theorem, which establishes a somewhat more general result.

Lemma 9 *In $\mathfrak{A}(s)$, $\varepsilon \xrightarrow{\vec{x}} t$ iff $\vec{x} \in L(t)$. Hence, the language accepted by state t is exactly $L(t)$.*

Proof. First, we show that if $\varepsilon \stackrel{\vec{x}}{\to} u$ then $\vec{x} \in L(u)$. This is done by induction on the length of $\vec{x}$. If $\vec{x} = \varepsilon$ the claim trivially follows. Now let $\vec{x} = \vec{y}a$ for some a. Assume that $\varepsilon \stackrel{\vec{x}}{\to} u$. Then there is t such that $\varepsilon \stackrel{\vec{y}}{\to} t \stackrel{a}{\to} u$. By inductive assumption, $\vec{y} \in L(t)$, and by definition of the transition relation, $L(t)a \subseteq L(u)$. Whence the claim follows. Now for the converse, the claim that if $\vec{x} \in L(u)$ then $\varepsilon \stackrel{\vec{x}}{\to} u$. Again we show this by induction on the length of $\vec{x}$. If $\vec{x} = \varepsilon$ we are done. Now let $\vec{x} = \vec{y}a$ for some $a \in A$. We have to show that there is a $t \in s^\dagger$ such that $ta \subseteq u$. For then by inductive assumption, $\varepsilon \stackrel{\vec{y}}{\to} t$, and so $\varepsilon \stackrel{\vec{y}a}{\to} u$, by definition of the transition relation.

Now for the remaining claim: if $\vec{y}a \in L(u)$ then there is a $t \in s^\dagger$ such that $\vec{y} \in L(t)$ and $L(ta) \subseteq L(u)$. Again induction this time on u. Notice right away that $u^\dagger \subseteq s^\dagger$, a fact that will become useful. Case 1. $u = b$ for some letter. Clearly, $\varepsilon \in s^\dagger$, and putting $t := \varepsilon$ will do. Case 2. $u = u_1 \cup u_2$. Then u_1 and u_2 are both in $s^\dagger$. Now, suppose $\vec{y}a \in L(u_1)$. By inductive hypothesis, there is t such that $L(ta) \subseteq L(u_1) \subseteq L(u)$, so the claim follows. Similarly if $\vec{y}a \in u_2$. Case 3. $u = u_1u_2$. Subcase 2a. $\vec{y} = \vec{y}_1\vec{y}_2$, with $\vec{y}_1 \in L(u_1)$ and $\vec{y}_2 \in L(u_2)$. Now, by inductive hypothesis, there is a t_2 such that $L(t_2a) \subseteq L(u_2)$. Then $t := u_1t_2$ is the desired term. Since it is in $u^\dagger$, it is also in $s^\dagger$. And $L(ta) = L(u_1t_2a) \subseteq L(u_1u_2) = L(u)$. Case 4. $u = u_1^*$. Suppose $\vec{y}a \in L(u)$. Then $\vec{y}$ has a decomposition $\vec{z}_0\vec{z}_1 \cdots \vec{z}_{n-1}\vec{v}$ such that $\vec{z}_i \in L(u_1)$ for all $i < n$, and also $\vec{v}a$ is in $L(u_1)$. By inductive hypothesis, there is a t_1 such that $L(t_1a) \subseteq L(u_1)$, and $\vec{v} \in L(t_1)$. And $\vec{z}_0\vec{z}_1 \cdots \vec{z}_{n-1} \in L(u)$. Now put $t := ut_1$. This has the desired properties. □

Now, as a consequence, if L is regular, so is L^p. Namely, take the automaton $\mathfrak{A}(s)$, where s is a regular term of L. Now change this automaton to make *every* state accepting. This defines a new automaton which accepts every string that falls under some $t \in s^\dagger$, by the previous results. Hence, it accepts all prefixes of string from L.

We discuss an application. Syllables of a given language are subject to certain conditions. One of the most famous constraints (presumed to be universal) is the **sonoricity hierarchy**. It states that the sonoricity of phonemes in a syllable must rise until the nucleus, and then fall. The nucleus contains the sounds of highest sonoricity (which do not have to be vowels). The rising part is called the **onset** and the falling part the **rhyme**. The sonoricity hierarchy translates into a finite state automaton as follows. It has states $\langle o, i\rangle$, and $\langle r, i\rangle$, where i is a number smaller

than 10. The transitions are of the form $q_0 \stackrel{a}{\to} \langle o, i\rangle$ if a has sonoricity i, $q_0 \stackrel{a}{\to} \langle r, i\rangle$ if a has sonoricity i $\langle o, i\rangle \stackrel{a}{\to} \langle o, j\rangle$ if a has sonoricity $j \geq i$; $\langle o, i\rangle \stackrel{a}{\to} \langle r, j\rangle$ if a has sonoricity $j > i$, $\langle r, i\rangle \stackrel{a}{\to} \langle r, j\rangle$ where a has sonoricity $j \leq i$. All states of the form $\langle r, i\rangle$ are accepting. (This accounts for the fact that a syllable must have a rhyme. It may lack an onset, however.) Refinements of this hierarchy can be implemented as well. There are also language specific constraints, for example, that there is no syllable that begins with [ŋ] in English. Moreover, only vowels, nasals and laterals may be nuclear. We have seen above that a conjunction of conditions which each are regular also is a regular condition. Thus, effectively (this has proved to be correct over and over) phonological conditions on syllable and word structure are regular.

13 Complexity and Minimal Automata

In this section we shall look the problem of recognizing a string by an automaton. Even though computers are nowadays very fast, it is still possible to reach the limit of their capabilities very easily, for example by making a simple task overly complicated. Although finite automata seem very easy at first sight, to make the programs run as fast as possible is a complex task and requires sophistication.

Given an automaton $\mathfrak{A}$ and a string $\vec{x} = x_0 x_1 \dots x_{n-1}$, how long does it take to see whether or not $\vec{x} \in L(\mathfrak{A})$? Evidently, the answer depends on both $\mathfrak{A}$ and $\vec{x}$. Notice that $\vec{x} \in L(\mathfrak{A})$ if and only if there are $q_i, i < n + 1$ such that

$$i_0 = q_0 \stackrel{x_0}{\to} q_1 \stackrel{x_1}{\to} q_2 \stackrel{x_2}{\to} q_2 \dots \stackrel{x_{n-1}}{\to} x_n \in F \tag{123}$$

To decide whether or not $q_i \stackrel{x_i}{\to} q_{i+1}$ just takes constant time: it is equivalent to $\langle q_i, x_i, q_{i+1}\rangle \in \delta$. The latter is a matter of looking up δ. Looking up takes time logarithmic in the size of the input. But the input is bounded by the size of the automaton. So it take roughly the same amount all the time.

A crude strategy is to just go through all possible assignments for the q_i and check whether they satisfy (**??**). This requires checking up to $|A^n|$ many assignments. This suggests that the time required is exponential in the length of the string. In fact, far more efficient techniques exist. What we do is the following. We start with i_0. In Step 1 we collect all states q_1 such that $i_0 \stackrel{x_0}{\to} q_1$. Call this the set H_1. In Step 2 we collect all states q_2 such that there is a $q_1 \in H_1$ and $q_1 \stackrel{x_1}{\to} q_2$.

In Step 3 we collect into H_3 all the q_3 such that there exists a $q_2 \in H_2$ such that $q_2 \stackrel{x_3}{\to} q_3$. And so on. It is easy to see that

$$i_0 \stackrel{x_0 x_1 \ldots x_j}{\longrightarrow} q_{j+1} \text{ iff } q_{j+1} \in H_{j+1} \tag{124}$$

Hence, $\vec{x} \in L(\mathfrak{A})$ iff $H_n \cap F \neq \emptyset$. For then there exists an accepting state in H_n. Here each step takes time quadratic in the number of states: for given H_j, we compute H_{j+1} by doing Q many lookups for every $q \in H_j$. However, this number is basically bounded for given $\mathfrak{A}$. So the time requirement is down to a constant depending on $\mathfrak{A}$ times the length of $\vec{x}$. This is much better. However, in practical terms this is still not good enough. Because the constant it takes to compute a single step is too large. This is because we recompute the transition H_j to H_{j+1}. If the string is very long, this means that we recompute the same problem over and over. Instead, we can precompute all the transitions. It turns out that we can define an automaton in this way that we can use in the place of $\mathfrak{A}$.

Definition 10 *Let $\mathfrak{A} = \langle A, Q, i_0, F, \delta\rangle$ be an automaton. Put $F^\wp := \{U \subseteq Q : U \cap F \neq \emptyset\}$. And let*

$$\delta^\wp = \{\langle H, a, J\rangle : \textit{ for all } q \in H \textit{ there is } q' \in J\textit{: } q \stackrel{a}{\to} q'\} \tag{125}$$

Then

$$\mathfrak{A}^\wp = \langle A, \wp(Q), \{i_0\}, F^\wp, \delta^\wp\rangle \tag{126}$$

*is called the **exponential of** $\mathfrak{A}$.*

Definition 11 *An automaton is **deterministic** if for all $q \in Q$ and $a \in A$ there is at most one $q' \in Q$ such that $q \stackrel{a}{\to} q'$.*

It is clear that for a deterministic and total automaton, all we have to do is to look up the next state in (**??**), which exists and is unique. For these automata, recognizing the language is linear in the string.

Theorem 12 *For every automaton $\mathfrak{A}$, the exponential $\mathfrak{A}^\wp$ is total and deterministic. Moreover, $L(\mathfrak{A}^\wp) = L(\mathfrak{A})$.*

So, the recipe to attack the problem '$\vec{x} \in L(\mathfrak{A})$?' is this: first compute $\mathfrak{A}^{\wp}$ and then check '$\vec{x} \in L(\mathfrak{A})$?'. Since the latter is deterministic, the time needed is actually linear in the length of the string, and logarithmic in the size of $\wp(Q)$, the state set of $\mathfrak{A}^{\wp}$. Hence, the time is linear also in $|Q|$.

We mention a corollary of Theorem **??**.

Theorem 13 *If $L \subseteq A^*$ is regular, so is $A^* - L$.*

Proof. Let L be regular. Then there exists a total deterministic automaton $\mathfrak{A} = \langle A, Q, i_0, F, \delta\rangle$ such that $L = L(\mathfrak{A})$. Now let $\mathfrak{B} = \langle A, Q, i_0, Q - F, \delta\rangle$. Then it turns out that $i_0 \xrightarrow{\vec{x}}_{\mathfrak{B}} q$ if and only if $i_0 \xrightarrow{\vec{x}}_{\mathfrak{A}} q$. Now, $\vec{x} \in L(\mathfrak{B})$ if and only if there is a $q \in Q - F$ such that $i_0 \xrightarrow{\vec{x}} q$ if and only if there is no $q \in F$ such that $i_0 \xrightarrow{\vec{x}} q$ if and only if $\vec{x} \notin L(\mathfrak{A}) = L$. This proves the claim. □

Now we have reduced the problem to the recognition by some deterministic automaton, we may still not have done the best possible. It may turn out, namely, that the automaton has more states than are actually necessary. Actually, there is no limit on the complexity of an automaton that recognizes a given language, we can make it as complicated as we want! The art, as always, is to make it simple.

Definition 14 *Let L be a language. Given a string $\vec{x}$, let $[\vec{x}]_L = \{\vec{y} : \vec{x}^\frown\vec{y} \in L\}$. We also write $\vec{x} \sim_L \vec{y}$ if $[\vec{x}]_L = [\vec{y}]_L$. The **index** of L is the number of distinct sets $[\vec{x}]_L$.*

Here is an example. The language $L = \{\mathsf{ab}, \mathsf{ac}, \mathsf{bc}\}$ has the following index sets:

$$\begin{array}{lcl} [\varepsilon]_L & = & \{\mathsf{ab}, \mathsf{ac}, \mathsf{bc}\} \\ [\mathsf{a}]_L & = & \{\mathsf{b}, \mathsf{c}\} \\ [\mathsf{b}]_L & = & \{\mathsf{c}\} \\ [\mathsf{c}]_L & = & \varnothing \\ [\mathsf{ab}]_L & = & \{\varepsilon\} \end{array} \tag{127}$$

Let us take a slightly different language $M = \{\mathsf{ab}, \mathsf{ac}, \mathsf{bc}, \mathsf{bb}\}$.

$$\begin{array}{lcl} [\varepsilon]_M & = & \{\mathsf{ab}, \mathsf{ac}, \mathsf{bb}, \mathsf{bc}\} \\ [\mathsf{a}]_M & = & \{\mathsf{b}, \mathsf{c}\} \\ [\mathsf{c}]_M & = & \varnothing \\ [\mathsf{ab}]_M & = & \{\varepsilon\} \end{array} \tag{128}$$

It is easy to check that $[\mathtt{b}]_M = [\mathtt{a}]_M$. We shall see that this difference means that there is an automaton checking M can based on less states than any automaton checking membership in L.

Given two index set I and J, put $I \stackrel{a}{\to} J$ if and only if $J = a\backslash I$. This is well-defined. For let $I = [\vec{x}]_L$. Then suppose that $\vec{x}a$ is a prefix of an accepted string. Then $[\vec{x}a]_L = \{\vec{y} : \vec{x}a\vec{y} \in L\} = \{\vec{y} : a\vec{y} \in I\} = a\backslash I$. This defines a deterministic automaton with initial element L. Accepting sets are those which contain ε. We call this the **index automaton** and denote it by $\mathfrak{I}(L)$. (Often it is called the **Myhill-Nerode automaton**.)

Theorem 15 (Myhill-Nerode) *$L(\mathfrak{I}(L)) = L$.*

Proof. By induction on $\vec{x}$ we show that $L \stackrel{\vec{x}}{\to} I$ if and only if $[\vec{x}]_L = I$. If $\vec{x} = \varepsilon$ the claim reads $L = L$ if and only if $[\varepsilon]_L = L$. But $[\varepsilon]_L = L$, so the claim holds. Next, let $\vec{x} = \vec{y}a$. By induction hypothesis, $L \stackrel{\vec{y}}{\to} J$ if and only if $[\vec{y}]_L = I$. Now, $J \stackrel{a}{\to} J/a = [\vec{y}a]_L$. So, $L \stackrel{\vec{x}}{\to} J/a = [\vec{x}]_L$, as promised.

Now, $\vec{x}$ is accepted by $\mathfrak{I}(L)$ if and only if there is a computation from L to a set $[\vec{y}]_L$ containing ε. By the above this is equivalent to $\varepsilon \in [\vec{x}]_L$, which means $\vec{x} \in L$. □

Given an automaton $\mathfrak{A}$ and a state q put

$$[q] := \{\vec{x} : \text{ there is } q' \in F\colon q \stackrel{\vec{x}}{\to} q'\} \tag{129}$$

It is easy to see that for every q there is a string $\vec{x}$ such that $[q] \subseteq [\vec{x}]_L$. Namely, let $\vec{x}$ be such that $i_0 \stackrel{\vec{x}}{\to} q$. Then for all $\vec{y} \in [q]$, $\vec{x}\vec{y} \in L(\mathfrak{A})$, by definition of $[q]$. Hence $[q] \subseteq [\vec{x}]_L$. Conversely, for every $[\vec{x}]_L$ there must be a state q such that $[q] \subseteq [\vec{x}]_L$. Again, q is found as a state such that $i_0 \stackrel{\vec{x}}{\to} q$. Suppose now that $\mathfrak{A}$ is deterministic and total. Then for each string $\vec{x}$ there is exactly one state $[q]$ such that $[q] \subseteq [\vec{x}]_L$. Then obviously $[q] = [\vec{x}]_L$. For if $\vec{y} \in [\vec{x}]_L$ then $\vec{x}\vec{y} \in L$, whence $i_0 \stackrel{\vec{x}\vec{y}}{\longrightarrow} q' \in F$ for some q'. Since the automaton is deterministic, $i_0 \stackrel{\vec{x}}{\to} q \stackrel{\vec{y}}{\to} q'$, whence $\vec{y} \in [q]$.

It follows now that the index automaton is the smallest deterministic and total automaton that recognizes the language. The next question is: how do we make that automaton? There are two procedures; one starts from a given automaton,

and the other starts from the regular term. Given an automaton, we know how to make a deterministic total automaton by using the exponentiation. On the other hand, let $\mathfrak{A}$ by an automaton. Call a relation $\sim$ a **net** if

❶ from $q \sim q'$ and $q \stackrel{a}{\to} r$, $q' \stackrel{a}{\to} r'$ follows $q' \sim r'$, and

❷ if $q \in F$ and $q \sim q'$ then also $q' \in F$.

A net induces a partition of the set of states into sets of the form $[q]_\sim = \{q' : q' \sim q\}$. In general, a **partition** of Q is a set Π of nonempty subsets of Q such that any two $S, S' \in \Pi$ which are distinct are disjoint; and every element is in one (and therefore only one) member of Π.

Given a net $\sim$, put

$$\begin{aligned} [q]_\sim &:= \{q' : q \sim q'\} \\ Q/\sim &:= \{[q]_\sim : q \in Q\} \\ F/\sim &:= \{[q]_\sim : q \in F\} \\ [q']_\sim \in \delta/\sim([q]_\sim, a) &\Leftrightarrow \text{there is } r \sim q' : r \in \delta(q, a) \\ \mathfrak{A}/\sim &:= \langle A, Q/\sim, [i_0]_\sim, F/\sim, \delta/\sim\rangle \end{aligned} \tag{130}$$

Lemma 16 $L(\mathfrak{A}/\sim) = L(\mathfrak{A})$.

Proof. By induction on the length of $\vec{x}$ the following can be shown: if $q \sim q'$ and $q \stackrel{\vec{x}}{\to} r$ then there is a $r' \sim r$ such that $q' \stackrel{\vec{x}}{\to} r'$. Now consider $\vec{x} \in L(\mathfrak{A}/\sim)$. This means that there is a $[q]_\sim \in F/\sim$ such that $[i_0]_\sim \stackrel{\vec{x}}{\to} [q]_\sim$. This means that there is a $q' \sim q$ such that $i_0 \stackrel{\vec{x}}{\to} q'$. Now, as $q \in F$, also $q' \in F$, by definition of nets. Hence $\vec{x} \in L(\mathfrak{A})$. Conversely, suppose that $\vec{x} \in L(\mathfrak{A})$. Then $i_0 \stackrel{\vec{x}}{\to} q$ for some $q \in F$. Hence $[i_0]_\sim \stackrel{\vec{x}}{\to} [q]_\sim$, by an easy induction. By definition of $\mathfrak{A}/\sim$, $[q]_\sim$ is an accepting state. Hence $\vec{x} \in L(\mathfrak{A}/\sim)$. □

All we have to do next is to fuse together all states which have the same index. Therefore, we need to compute the largest net on $\mathfrak{A}$. The is done as follows. In the first step, we put $q \sim_0 q'$ iff $q, q' \in F$ or $q, q' \in Q - F$. This need not be a net.

Inductively, we define the following:

(131) $q \sim_{i+1} q' :\Leftrightarrow q \sim_i q'$ and for all $a \in A$, for all $r \in Q$:
if $q \xrightarrow{a} r$ there is $r' \sim_i r$ such that $q' \xrightarrow{a} r'$
if $q' \xrightarrow{a} r$ there is $r' \sim_i r$ such that $q' \xrightarrow{a} r$

Evidently, $\sim_{i+1} \subseteq \sim_i$. Also, if $q \sim_{i+1} q'$ and $q \in F$ then also $q' \in F$, since this already holds for $\sim_0$. Finally, if $\sim_{i+1} = \sim_i$ then $\sim_i$ is a net. This suggests the following recipe: start with $\sim_0$ and construct $\sim_i$ one by one. If $\sim_{i+1} = \sim_i$, stop the construction. It takes only finitely many steps to compute this and it returns the largest net on an automaton.

Definition 17 *Let $\mathfrak{A}$ be a finite state automaton. $\mathfrak{A}$ is called **refined** if the only net on it is the identity.*

Theorem 18 *Let $\mathfrak{A}$ and $\mathfrak{B}$ be deterministic, total and refined and every state in each of the automata is reachable. Then if $L(\mathfrak{A}) = L(\mathfrak{B})$, the two are isomorphic.*

Proof. Let $\mathfrak{A}$ be based on the state set $Q^{\mathfrak{A}}$ and $\mathfrak{B}$ on the state set $Q^{\mathfrak{B}}$. For $q \in Q^{\mathfrak{A}}$ write $I(q) := \{\vec{x} : q \xrightarrow{\vec{x}} r \in F\}$, and similarly for $q \in Q^{\mathfrak{B}}$. Clearly, we have $I(i_0^{\mathfrak{A}}) = I(i_0^{\mathfrak{B}})$, by assumption. Now, let $q \in Q^{\mathfrak{A}}$ and $q \xrightarrow{a} r$. Then $I(r) = a\backslash I(q)$. Hence, if $q' \in Q^{\mathfrak{B}}$ and $q' \xrightarrow{a} r'$ and $I(q) = I(q')$ then also $I(r) = I(r')$. Now we construct a map $h : Q^{\mathfrak{A}} \to Q^{\mathfrak{B}}$ as follows. $h(i_0^{\mathfrak{A}}) := i_0^{\mathfrak{B}}$. If $h(q) = q'$, $q \xrightarrow{a} r$ and $q' \xrightarrow{a} r'$ then $h(r) := r'$. Since all states are reachable in $\mathfrak{A}$, h is defined on all states. This map is injective since $I(h(q)) = I(q)$ and $\mathfrak{A}$ is refined. (Every homomorphism induces a net, so if the identity is the only net, h is injective.) It is surjective since all states in $\mathfrak{B}$ are reachable and $\mathfrak{B}$ is refined. □

Thus the recipe to get a minimal automaton is this: get a deterministic total automaton for L and refine it. This yields an automaton which is unique up to isomorphism.

The other recipe is dual to Lemma **??**. Put

$$0^\ddagger := \varnothing \tag{132a}$$

$$a^\ddagger := \{\varepsilon, a\} \tag{132b}$$

$$\varepsilon^\ddagger := \{\varepsilon\} \tag{132c}$$

$$(s \cup t)^\ddagger := s^\ddagger \cup t^\ddagger \tag{132d}$$

$$(st)^\ddagger := \{ut : u \in s^\ddagger\} \cup t^\ddagger \tag{132e}$$

$$(s^*)^\ddagger := s^\ddagger \cup \{us^* : u \in s^\ddagger\} \tag{132f}$$

Given a regular expression s, we can effectively compute the sets $[\vec{x}]_L$. They are either 0 or of the form t for $t \in s^\ddagger$. The start symbol is s, and the accepting states are of the form $t \in s^\ddagger$, where $\varepsilon \in t$. However, beware that it is not clear a priori for any two given terms t, u whether or $L(t) = L(u)$. So we need to know how we can effectively decide this problem. Here is a recipe. Construct an automaton $\mathfrak{A}$ such that $L(\mathfrak{A}) = L(t)$ and an automaton $L(\mathfrak{B}) = A^* - L(u)$. We can assume that they are deterministic. Then $\mathfrak{A} \times \mathfrak{B}$ recognizes $L(t) \cap (A^* - L(u))$. This is empty exactly when $L(t) \subseteq L(u)$. Dually we can construct an automaton that recognizes $L(u) \cap (A^* - L(t))$, which is empty exactly when $L(u) \subseteq L(t)$. So, everything turns on the following question: can we decide for any given automaton $\mathfrak{A} = \langle A, Q, i_0, F, \delta \rangle$ whether or not $L(\mathfrak{A})$ is empty? The answer is simple: this is decidable. To see how this can be done, notice that $L(\mathfrak{A}) \neq \varnothing$ iff there is a word $\vec{x}$ such that $i_0 \stackrel{\vec{x}}{\to} q \in F$. It is easy to see that if there is a word, then there is a word whose length is $\leq |Q|$. Now we just have to search through all words of this size.

Theorem 19 *The following problems are decidable for given regular terms t, u: '$L(t) = \varnothing$', '$L(t) \subseteq L(u)$' and '$L(t) = L(u)$'.* □

It now follows that the index machine can effectively be constructed. The way to do this is as follows. Starting with s, we construct the machine based on the $t \in s^\ddagger$. Next we compute for given t' whether the equivalence $L(t) = L(t')$ holds for some $t \neq t'$. Then we add a new state t''. Every arc into t'' is an arc that goes into t or into t'; every arc leaving t'' is an arc that leaves t or t'. Erase t and t'.

Theorem 20 *L is regular if and only if it has only finitely many index sets.*

14 Digression: Time Complexity

The algorithm that decides whether or not two automata accept the same language, or whether the language accepted by an automaton is empty, takes exponential time. All it requires us to do is to look through the list or words that are of length $\leq n$ and check whether they are accepted. If A has α many letters, there are α^n words of length n, and

$$1 + \alpha + \alpha^2 + \ldots + \alpha^n = (\alpha^{n+1} - 1)/(\alpha - 1) \tag{133}$$

many words of length $\leq n$. For simplicity we assume that $\alpha = 2$. Then $(2^{n+1} - 1)/(2 - 1) = 2^{n+1} - 1$. Once again, let us simplify a little bit. Say that the number of steps we need is 2^n. Here is how fast the number of steps grows.

(134)

n	2^n	n	2^n
1	2	11	2048
2	4	12	4096
3	8	13	8192
4	16	14	16384
5	32	15	32768
6	64	16	65536
7	128	17	131072
8	256	18	262144
9	512	19	524288
10	1024	20	1048576

Suppose that your computer is able to compute 1 million steps in a second. Then for $n = 10$ the number of steps is 1024, still manageable. It only takes a millisecond (1/1000 of a second). For $n = 20$, it is 1,048,576; this time you need 1 second. However, for each 10 you add in size, the number of steps multiplies by more than 1000! Thus, for $n = 30$ the time needed is 18 minutes. Another 10 added and you can wait for months already. Given that reasonable applications in natural language require several hundreds of states, you can imagine that your computer might not even be able to come up with an answer in your lifetime.

Thus, even with problems that seem very innocent we may easily run out of time. However, it is not necessarily so that the operations we make the computer perform are really needed. Maybe there is a faster way to do the same job. Indeed, the problem just outlined can be solved much faster than the algorithm just shown

would lead one to believe. Here is another algorithm: call a state q n–**reachable** if there is a word of length $\leq n$ such that $i_0 \xrightarrow{\vec{x}} q$; call it **reachable** if it is n–reachable for some $n \in \omega$. $L(\mathfrak{A})$ is nonempty if some $q \in F$ is reachable. Now, denote by R_n the set of n–reachable states. $R_0 := \{i_0\}$. In the nth step we compute R_{n+1} by taking all successors of points in R_n. If $R_{n+1} = R_n$ we are done. (Checking this takes $|Q|$ steps. Alternatively, we need to iterate the construction at most $|Q| - 1$ times. Because if at each step we add some state, then R_n grows by at least 1, so this can happen only $|Q| - 1$ times, for then R_n has at least $|Q|$ many elements. On the other hand, it is a subset of Q, so it can never have more elements.) Let us now see how much time we need. Each step takes $R_n \times |A| \leq |Q| \times |A|$ steps. Since $n \leq |Q|$, we need at most $|Q| - 1$ steps. So, we need $c \cdot |A||Q|^2$ steps for some constant c.

Consider however an algorithm that takes n^2 steps.

(135)

n	n^2	n	n^2
1	1	11	121
2	4	12	144
3	9	13	169
4	16	14	196
5	25	15	225
6	36	16	256
7	49	17	289
8	64	18	324
9	81	19	361
10	100	20	400

For $n = 10$ it takes 100 steps, or a tenth of a millisecond, for $n = 20$ it takes 400, and for $n = 30$ only 900, roughly a millisecond. Only if you double the size of the input, the number of steps quadruple. Or, if you want the number of steps to grow by a factor 1024, you have to multiply the length of the input by 32! An input of even a thousand creates a need of only one million steps and takes a second on our machine.

The algorithm described above is not optimal. There is an algorithm that is even faster: It goes as follows. We start with $R_{-1} := \varnothing$ and $S_0 = \{q_0\}$. Next, S_1 is the set of successors of S_0 minus R_0, and $R_0 := S_0$. In each step, we have two sets, S_i and R_{i-1}. R_{i-1} is the set of $i-1$–reachable points, and S_i is the set of points that can be reached from it it one step, but which are not in R_{i-1}. In the next step

we let S_{i+1} be the set of successors of S_i which are not in $R_i := R_{i-1} \cup S_i$. The advantage of this algorithm is that it does not recompute the successors of a given point over and over. Instead, only the successors of point that have recently been added are calculated. It is easily seen that this algorithm computes the successors of a point only once. Thus, this algorithm is not even quadratic but linear in $|Q|$! And if the algorithm is linear — that is much better. Actually, you cannot get less than that if the entire input matters. For the machine needs to take a look at the input — and this take slinear time at least.

It is not always the case that algorithms can be made as fast as this one. The problem of satisfiability of a given propositional formula is believed to take exponential time. More exactly, it is in **NP** — though it may take polynomial time in many cases. Regardless whether an algorithm takes a lot of time or not it is wise to try an reduce the number of steps that it actually takes. This can make a big difference when the application has to run on real life examples. Often, the best algorithms are not so difficult to understand — one just has to find them. Here is another one. Suppose you are given a list of numbers. Your task is to sort the list in ascending order as fast as possible. Here is the algorithm that one would normally use: scan the list for the least number and put it at the beginning of a new list; then scan the remainder for the least number and put that behind the number you already have, and so on. In step n you have you original list L, reduced by $n-1$ elements, and a list L' containing $n-1$ elements. To find the minimal element in L you do the following: you need two memory cells, C and P. C initially contains the first element of the list and $P := 1$. Now take the second element and see whether it is smaller than the content of C; if so, you put it into C, and let P be the number of the cell; if not, you leave C and P as is. You need to do $|L|-1$ comparisons, as you have to go through the entire list. When you are done, P tells you which element to put into M. Now you start again. The number of comparisons is overall

$$(|L|-1) + (|L|-2) + \ldots + 2 + 1 = |L| \cdot (|L|-1)/2 \tag{136}$$

This number grows quadratically. Not bad, but typically lists are very long, so we should try the best we can.

Here is another algorithm. Divide the list into blocks of two. (There might be a remainder of one element, but that does no harm.) We order these blocks. This takes just one comparison between the two elements of the block. In the next step we merge two blocks of two into one block of four. In the third step we merge

two blocks of four into one block of eight, and so on. How many comparisons are needed to merge two ordered lists M and N of length m and n, respectively, into a list L of length $m + n$? The answer is: $m + n - 1$. The idea is as follows. We take the first elements, M_0 and N_0. Then L_0 is the smaller of the two, which is then removed from its list. The next element is again obtained by comparing the first elements of the lists, and so on. For example, let $M = [1, 3, 5]$ and $N = [2, 4, 5]$. We compare the first two elements. The smaller one is put into L: the lists are now

(137) $M = [3, 5], N = [2, 4, 5], L = [1]$

Now, we compare the first elements of M and N. The smallest element is 2 and is put into L:

(138) $M = [3, 5], N = [4, 5], L = [1, 2]$

Now, this is how the algorithm goes on:

(139) $M = [5], N = [4, 5], L = [1, 2, 3]$

(140) $M = [5], N = [5], L = [1, 2, 3, 4]$

(141) $M = [], N = [5], L = [1, 2, 3, 4, 5]$

(142) $M = [], N = [], L = [1, 2, 3, 4, 5, 5]$

Each time we put an element into L we need just one comparison, except for the last element, which can be put in without further checking. If we want to avoid repetitions then we need to check each element against the last member of the list before putting it in (this increases the number of checks by $n + m - 1$).

In the first step we have $n/2$ many blocks, and $n/4$ many comparisons are being made to order them. The next step takes $3n/4$ comparisons, the third step needs $7n/8$, and so on. Let us round the numbers somewhat: each time we need certainly less than n comparisons. How often do we have to merge? This number is $\log_2 n$. This is the number x such that $2^x = n$. So we need in total $n \log_2 n$ many steps. We show this number in comparison to n and n^2.

(143)

	2^0	2^4	2^8	2^{12}	2^{16}
n	1	16	256	4096	65536
$n \log_2 n$	0	64	2048	59152	1048576
$n^2/2$	1/2	128	32768	8388608	2147483648

Consider again your computer. On an input of length 65536 (= 2^{16}) it takes one second under the algorithm just described, while the naive algorithm would require it run for 2159 seconds, which is more than half an hour.

In practice, one does not want to spell out in painful detail how many steps an algorithm consumes. Therefore, simplifying notation is used. One writes that a problem is in $O(n)$ if there is a constant C such that from some n_0 on for an input of length n the algorithm takes $C \cdot n$ steps to compute the solution. (One says that the estimate holds for 'almost all' inputs if it holds only from a certain point onwards.) This notation makes sense also in view of the fact that it is not clear how much time an individual step takes, so that the time consumption cannot not really be measured in seconds (which is what is really of interest for us). If tomorrow computers can compute twice as fast, everything runs in shorter time. Notice that $O(bn + a) = O(bn) = O(n)$. It is worth understanding why. First, assume that $n \geq a$. Then $(b+1)n \geq bn+n \geq bn+a$. This means that for almost all n: $(b+1)n \geq bn+a$. Next, $O((b+1)n) = O(n)$, since $O((b+1)n)$ effectively means that there is a constant C such that for almost all n the complexity is $\leq C(b+1)n$. Now put $D := C(b+1)$. Then there is a constant (namely D) such that for almost all n the complexity is $\leq Dn$. Hence the problem is in $O(n)$.

Also $O(cn^2 + bn + a) = O(n^2)$ and so on. In general, the highest exponent wins by any given margin over the others. Polynomial complexity is therefore measured only in terms of the leading exponent. This makes calculations much simpler.

15 Finite State Transducers

Finite state transducers are similar to finite state automata. You think of them as finite state automata that leave a trace of their actions in the form of a string. However, the more popular way is to think of them as translation devices with finite memory. A **finite state transducer** is a sextuple

(144) $\mathfrak{T} = \langle A, B, Q, i_0, F, \delta \rangle$

where A and B are alphabets, Q a finite set (the set of **states**), i_0 the **initial state**, F the set of **final states** and

(145) $\delta \subseteq \wp(A_\varepsilon \times Q \times B_\varepsilon \times Q)$

(Here, $A_\varepsilon := A \cup \{\varepsilon\}$, and ${}_\varepsilon := B \cup \{\varepsilon\}$.) We write $q \xrightarrow{a:b} q'$ if $\delta(a, q, b, q') \in \delta$. We say in this case that $\mathfrak{T}$ makes a transition from q to q' given input a, and that it outputs b. Again, it is possible to define this notation for strings. So, if $q \xrightarrow{\vec{u}:\vec{v}} q'$ and $q' \xrightarrow{\vec{x}:\vec{y}} q''$ then we write $q \xrightarrow{\vec{u}\vec{x}:\vec{v}\vec{y}} q''$.

It is not hard to show that a finite state transducer from A to B is equivalent to a finite state automaton over $A \times B$. Namely, put

$$\mathfrak{A} := \langle A \times B, Q, i_0, F, \theta \rangle \tag{146}$$

where $q' \in \theta(q, \langle a, b \rangle)$ iff $\langle a, q, b, q' \rangle \in \delta$. Now define

$$\langle \vec{u}, \vec{v} \rangle \cdot \langle \vec{x}, \vec{y} \rangle := \langle \vec{u}\vec{x}, \vec{x}\vec{y} \rangle \tag{147}$$

Then one can show that $q \xrightarrow{\vec{x}:\vec{y}} q'$ in $\mathfrak{T}$ if and only if $q \xrightarrow{\langle \vec{x}, \vec{y} \rangle} q'$ in $\mathfrak{A}$. Thus, the theory of finite state automata can be used here. Transducers have been introduced as devices that translate languages. We shall see many examples below. Here, we shall indicate a simple one. Suppose that the input is $A := \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$ and $B := \{\mathtt{0}, \mathtt{1}\}$. We want to translate words over A into words over B in the following way. a is translated by $\mathtt{00}$, b by $\mathtt{01}$, c by $\mathtt{10}$ and d by $\mathtt{11}$. Here is how this is done. The set of states is $\{0, 1, 2\}$. The initial state is 0 and the accepting states are $\{0\}$. The transitions are

$$\langle 0, \mathsf{a}, 1, \mathtt{0} \rangle, \langle 0, \mathsf{b}, 2, \mathtt{0} \rangle, \langle 0, \mathsf{c}, 1, \mathtt{1} \rangle, \langle 0, \mathsf{d}, 2, \mathtt{1} \rangle, \langle 1, \varepsilon, 0, \mathtt{0} \rangle, \langle 2, \varepsilon, 0, \mathtt{1} \rangle \tag{148}$$

This means the following. Upon input a, the machine enters state 1, and outputs the letter $\mathtt{0}$. It is not in an accepting state, so it has to go on. There is only one way: it reads no input, returns to state 0 and outputs the letter $\mathtt{0}$. Now it is back to where is was. It has read the letter a and mapped it to the word $\mathtt{00}$. Similarly it maps b to $\mathtt{01}$, c to $\mathtt{10}$ and d to $\mathtt{11}$.

Let us write $R_{\mathfrak{T}}$ for the following relation.

$$R_{\mathfrak{T}} := \{\langle \vec{x}, \vec{y} \rangle : i_0 \xrightarrow{\vec{x}:\vec{y}} q \in F\} \tag{149}$$

Then, for every word $\vec{x}$, put

$$R_{\mathfrak{T}}(\vec{x}) := \{\vec{y} : \vec{x}\, R_{\mathfrak{T}}\, \vec{y}\} \tag{150}$$

And for a set $S \subseteq A^*$

(151) $\quad R_{\mathfrak{T}}[S] := \{\vec{y} : \text{exists } \vec{x} \in S : \vec{x} R_{\mathfrak{T}} \vec{y}\}$

This is the set of all strings over B that are the result of translation via $\mathfrak{T}$ of a string in S. In the present case, notice that every string over A has a translation, but not every string over B is the translation of a string. This is the case if and only if it has even length.

The translation need not be unique. Here is a finite state machine that translates a into bc^*.

(152) $\quad A := \{\mathsf{a}\}, B := \{\mathsf{b}, \mathsf{c}\}, Q := \{0, 1\}, i_0 := 0, F := \{1\},$
$\delta := \{\langle 0, \mathsf{a}, 1, \mathsf{b}\rangle, \langle 1, \varepsilon, 1, \mathsf{c}\rangle\}$

This automaton takes as only input the word a. However, it outputs any of b, bc, bcc and so on. Thus, the translation of a given input can be highly underdetermined. Notice also the following. For a language $S \subseteq A^*$, we have the following.

(153) $$R_{\mathfrak{T}}[S] = \begin{cases} \mathsf{bc}^* & \text{if } \mathsf{a} \in S \\ \varnothing & \text{otherwise.} \end{cases}$$

This is because only the string a has a translation. For all words $\vec{x} \neq \mathsf{a}$ we have $R_{\mathfrak{T}}(\vec{x}) = \varnothing$.

The transducer can also be used the other way: then it translates words over B into words over A. We use the same machine, but now we look at the relation

(154) $\quad R^{\smile}_{\mathfrak{T}}(\vec{y}) := \{\vec{x} : \vec{x} R_{\mathfrak{T}} \vec{y}\}$

(This is the converse of the relation $R_{\mathfrak{T}}$.) Similarly we define

(155) $\quad R^{\smile}_{\mathfrak{T}}(\vec{y}) := \{\vec{x} : \vec{x} R_{\mathfrak{T}} \vec{y}\}$

(156) $\quad R^{\smile}_{\mathfrak{T}}[S] := \{\vec{x} : \text{exists } \vec{y} \in S : \vec{x} R_{\mathfrak{T}} \vec{y}\}$

Notice that there is a transducer $\mathfrak{U}$ such that $R_{\mathfrak{U}} = R^{\smile}_{\mathfrak{T}}$.

We quote the following theorem.

Theorem 21 (Transducer Theorem) *Let $\mathfrak{T}$ be a transducer from A to B, and let $L \subseteq A^*$ be a regular language. Then $R_{\mathfrak{T}}[L]$ is regular as well.*

Notice however that the transducer can be used to translate any language. In fact, the image of a context free language under $R_{\mathfrak{T}}$ can be shown to be context free as well.

Let us observe that the construction above can be generalized. Let A and B be alphabets, and f a function assigning each letter of A a word over B. We extend f to words over A as follows.

(157) $f(\vec{x} \cdot a) := \vec{x} \cdot f(a)$

The translation function f can be effected by a finite state machine in the following way. The initial state is i_0. On input a the machine goes into state q_a, outputs ε. Then it returns to i_0 in one or several steps, outputting $f(a)$. Then it is ready to take the next input. However, there are more complex functions that can be calculated with transducers.

16 Finite State Morphology

One of the most frequent applications of transducers is in morphology. Practically all morphology is finite state. This means the following. There is a finite state transducer that translates a gloss (= deep morphological analysis) into surface morphology. For example, there is a simple machine that puts English nouns into the plural. It has two states, 0, and 1; 0 is initial, 1 is final. The transitions are as follows (we only use lower case letters for ease of exposition).

(158) $\langle 0, \mathtt{a}, 0, \mathtt{a}\rangle, \langle 0, \mathtt{b}, 0, \mathtt{b}\rangle, \ldots, \langle 0, \mathtt{z}, 0, \mathtt{z}\rangle, \langle 0, \varepsilon, 1, \mathtt{s}\rangle.$

The machine takes an input and repeats it, and finally attaches `s`. We shall later see how we can deal with the full range of plural forms including exceptional plurals. We can also write a machine that takes a deep representation, such as `car` plus singular or `car` plus plural and outputs `car` in the first case and `cars` in the second. For this machine, the input alphabet has two additional symbols, say, ® and Ⓢ, and works as follows.

(159) $\langle 0, \mathtt{a}, 0, \mathtt{a}\rangle, \langle 0, \mathtt{b}, 0, \mathtt{b}\rangle, \ldots, \langle 0, \mathtt{z}, 0, \mathtt{z}\rangle, \langle 0, ®, 1, \varepsilon\rangle, \langle 0, Ⓢ, 1, \mathtt{s}\rangle.$

This machine accepts one ® or Ⓢ at the end and transforms it into ε in the case of ® and into `s` otherwise. As explained above we can turn the machine around.

Then it acts as a map from surface forms to deep forms. It will translate arm into arm® and arms into arm Ⓢ and arms®. The latter may be surprising, but the machine has no idea about the lexicon of English. It assumes that s can be either the sign of plural or the last letter of the root. Both cases arise. For example, the word bus has as its last letter indeed s. Thus, in one direction the machine synthesizes surface forms, and in the other direction it analyses them.

Now, let us make the machine more sophisticated. The regular plural is formed by adding es, not just s, when the word ends in sh: bushes, splashes. If the word ends in s then the plural is obtained by adding ses: busses, plusses. We can account for this as follows. The machine will take the input and end in three different states, according to whether the word ends in s, sh or something else.

(160)
$$\begin{array}{llll}
\langle 0,\mathtt{a},0,\mathtt{a}\rangle, & \langle 0,\mathtt{b},0,\mathtt{b}\rangle, & \ldots, & \langle 0,\mathtt{z},0,\mathtt{z}\rangle,\\
\langle 0,\mathtt{a},4,\mathtt{a}\rangle, & \ldots, & \langle 0,\mathtt{r},4,\mathtt{r}\rangle, & \langle 0,\mathtt{t},4,\mathtt{t}\rangle,\\
\ldots, & \langle 0,\mathtt{z},4,\mathtt{z}\rangle, & \langle 0,\mathtt{s},2,\mathtt{s}\rangle, & \langle 2,\mathtt{a},4,\mathtt{a}\rangle,\\
\ldots, & \langle 2,\mathtt{g},4,\mathtt{g}\rangle, & \langle 2,\mathtt{h},3,\mathtt{h}\rangle, & \langle 2,\mathtt{i},4,\mathtt{i}\rangle,\\
\ldots, & \langle 2,\mathtt{z},4,\mathtt{z}\rangle, & \langle 2,\varepsilon,3,\mathtt{s}\rangle, & \langle 3,\varepsilon,4,\mathtt{e}\rangle,\\
\langle 4,\text{®},1,\varepsilon\rangle, & \langle 4,\text{Ⓢ},1,\mathtt{s}\rangle. & &
\end{array}$$

This does not exhaust the actual spelling rules for English, but it should suffice. Notice that the machine, when turned around, will analyze busses correctly as busⓈ, and also as busses®. Once again, the mistake is due to the fact that the machine does not know that busses is no basic word of English. Suppose we want to implement that kind of knowledge into the machine. Then what we would have to do is write a machine that can distinguish an English word from a nonword. Such a machine probably requires very many states. It is probably no exaggeration to say that several hundreds of states will be required. This is certainly the case if we take into account that certain nouns form the plural differently: we only mention formulae (from formula, indices (from index), tableaux (from tableau), men, children, oxen, sheep, mice.

Here is another task. In Hungarian, case suffixes come in different forms. For example, the dative is formed by adding nak or nek. The form depends on the following factors. If the root contains a back vowel (a, o, u) then the suffix is nak; otherwise it is nek. The comitative suffix is another special case: when added, it becomes a sequence of consonant plus al or el (the choice of vowel depends in the same way as that of nak versus nek). The consonant is v if the root ends in a vowel; otherwise it is the same as the preceding one. (So: dob (‘drum’) dobnak

(‘for a drum’) dobbal (‘with a drum’); szeg (‘nail’) szegnek (‘for a nail’) and szeggel (‘with a nail’).) On the other hand, there are a handful of words that end in z (ez ‘this’, az ‘that’) where the final z assimilates to the following consonant (ennek ‘to this’), except in the comitative where we have ezzel ‘with this’. To write a finite state transducer, we need to record in the state two things: whether or not the root contained a back vowel, and what consonant the root ended in.

Plural in German is a jungle. First, there are many ways in which the plural can be formed: suffix s, suffix en, suffix er, Umlaut, which is the change (graphically) from a to ä, from o to ö and from u to ü of the last vowel; and combinations thereof. Second, there is no way to predict phonologically which word will take which plural. Hence, we have to be content with a word list. This means, translated into finite state machine, that we end up with a machine of several hundreds of states.

Another area where a transducer is useful is in writing conventions. In English, final y changes to i when a vowel is added: happy : happier, fly : flies. In Hungarian, the palatal sound [dj] is written gy. When this sound is doubled it becomes ggy and not, as one would expect, gygy. The word hegy should be the above rule become hegygyel, but the orthography dictates heggyel. (Actually, the spelling gets undone in hyphenation: you write hegy-<newline>gyel.) Thus, the following procedure suggests itself: we define a machine that regularizes the orthography by reversing the conventions as just shown. This machine translates heggyel into hegygyel. Actually, it is not necessary that gy is treated as a digraph. We can define a new alphabet in which gy is written by a single symbol. Next, we take this as input to a second machine which produces the deep morphological representations.

We close with an example from Egyptian Arabic. Like in many semitic languages, roots only consist of consonants. Typically, they have three consonants, for example ktb ‘to write’ and drs ‘to study’. To words are made by adding some material in front (prefixation), some material after (suffixation) and some material in between (infixation). Moreover, all these typically happen *at the same time*.

Let's look at the following list.

(161)

[katab]	'he wrote'	[daras]	'he studied'
[ʔamal]	'he did'	[naʿal]	'he copied'
[baktib]	'I write'	[badris]	'I study'
[baʔmil]	'I do'	[banʿil]	'I copy'
[iktib]	'write!'	[idris]	'study!'
[iʔmil]	'do!'	[inʿil]	'copy!'
[kaatib]	'writer'	[daaris]	'studier'
[ʔaamil]	'doer'	[naaʿil]	'copier'
[maktuub]	'written'	[madruus]	'studied'
[maʔmuu]	'done'	[manʿuul]	'copied'

Now, we want a transducer to translate `katab` into a sequence `ktb` plus 3rd, plus singular plus past. Similarly with the other roots. And it shall translate `baktib` into `ktb` plus 1st, plus singular plus present. And so on. The transducer will take the form CaCaC and translate it into CCC plus the markers 3rd, singular and past. (Obviously, one can reverse this and ask the transducer to spell out the form CCC plus 3rd, singular, past into CaCaC.)

17 Using Finite State Transducers

Transducers can be nondeterministic, and this nondeterminism we cannot always eliminate. One example is the following machine. It has two states, 0 and 1, 0 is initial, and 1 is accepting. It has the transitions $0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1$ and $1 \xrightarrow{\varepsilon:\mathsf{a}} 1$. This machine accepts a as input, together with any output string ba^n, $n \in \mathbb{N}$. What runs does this machine have? Even given input a there are infinitely many runs:

(162)
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$

In addition, given a specific output, sometimes there are several runs on that given pair. Here is an example. Again two states, 0 and 1, and the following transitions:

$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1$ and then $1 \xrightarrow{\varepsilon:\mathsf{b}} 1$ and $1 \xrightarrow{\mathsf{a}:\varepsilon} 1$. Now, for the pair `aaa` and `bbbb` there are several runs:

(163)
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$

(There are more runs that shown.) In the case of the machine just shown, the number of runs grows with the size of the input/output pair. There is therefore no hope of fixing the number of runs a priori.

Despite all this, we want to give algorithms to answer the following questions:

- ☞ Given a transducer $\mathfrak{T}$ and a pair $\vec{x} : \vec{y}$ of strings, is there a run of $\mathfrak{T}$ that accepts this pair? (That is to say: is $\langle \vec{x}, \vec{y} \rangle \in R_{\mathfrak{T}}$?)
- ☞ Given $\vec{x}$, is there a string $\vec{y}$ such that $\langle \vec{x}, \vec{y} \rangle \in R_{\mathfrak{T}}$?
- ☞ Given $\vec{x}$, is there a way to enumerate $R_{\mathfrak{T}}(\vec{x})$, that is to say, all $\vec{y}$ such that $\langle \vec{x}, \vec{y} \rangle \in R_{\mathfrak{T}}$?

Let us do these questions in turn. There are two ways to deal with nondeterminism: we follow a run to the end and backtrack on need (depth first analysis); or we store all partial runs and in each cycle try to extend each run if possible (breadth first search). Let us look at them in turn.

Let is take the second machine. Assume that the transitions are numbered:

(164) $t(0) = \langle 0, \mathsf{a}, 1, \mathsf{b} \rangle$

(165) $t(1) = \langle 1, \mathsf{a}, 1, \varepsilon \rangle$

(166) $t(2) = \langle 1, \varepsilon, 1, \mathsf{b} \rangle$

This numbering is needed to order the transitions. We start with the initial state 0. Let input `aaa` and output `bbbb` be given. Initially, no part of input or output is read. (Output is at present a bit of a misnomer because it is part of the input.) Now we want to look for a transition that starts at the state we are in, and matches

characters of the input and output strings. There is only one possibility: the transition $t(0)$. This results in the following: we are now in state 1, and one character from both strings has been read. The conjunction of facts (i) is in state 1, (ii) has read one character from in put, (iii) has read one character from output we call a **configuration**. We visualize this configuration by $1, \mathsf{a}|\mathsf{aa} : \mathsf{b}|\mathsf{bbb}$. The bar is placed right after the characters that have been consumed or *read*.

Next we face a choice: either we take $t(1)$ and consume the input a but no output letter, or we take $t(2)$ and consume the output b but no input letter. Faced with this choice, we take the transition with the least number, so we follow $t(1)$. The configuration is now this: $1, \mathsf{aa}|\mathsf{a} : \mathsf{b}|\mathsf{bbb}$. Now face the same choice again, to do either $t(1)$ or $t(2)$. We decide in the same way, choosing $t(1)$, giving $1, \mathsf{aaa}| : \mathsf{b}|\mathsf{bbb}$. Now, no more choice exists, and we can only proceed with $t(2)$, giving $1, \mathsf{aaa}| : \mathsf{bbb}|\mathsf{b}$. One last time we do $t(2)$, and we are done. This is exactly the first of the runs. Now, backtracking can be one for any reason. In this case the reason is: we want to find *another run*. To do that, we go back to the last point where there was an actual choice. This was the configuration $1, \mathsf{aa}|\mathsf{a} : \mathsf{b}|\mathsf{bbb}$. Here, we were able to choose between $t(1)$ and $t(2)$ and we chose $t(1)$. In this case we decide differently: now we take $t(2)$, giving $1, \mathsf{aa}|\mathsf{a} : \mathsf{bb}|\mathsf{bb}$. From this moment we have again a choice, but now we fall back on our typical recipe: choose $t(1)$ whenever possible. This gives the third run above. If again we backtrack, it is the last choice point within the run we are currently considering that we look at. Where we chose $t(1)$ we now choose $t(2)$, and we get this run:

(167) $$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$

Here is how backtracking continues to enumerate the runs:

(168)
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1$$

In general, backtracking works as follows. Every time we face a choice, we pick the available transition with the smallest number. If we backtrack we go back in the run to the point where there was a choice, and then we change the previously chosen next transition to the next up. From that point on we consistently choose the least numbers again. This actually enumerates all runs. It is sometimes a dangerous strategy since we need to guarantee that the run we are following actually terminates: this is no problem here, since every step consumes at least one character.

Important in backtracking is that it does not give the solutions in one blow: it gives us one at a time. We need to memorize only the last run, and then backtracking gives us the next if there is one. (If there is none, it will tell us. Basically, there is none if going back we cannot find any point at which there was a choice (because you can never choose lower number than you had chosen).)

Now let us look at the other strategy. It consists in remembering partial runs, and trying to complete them. A partial run on a pair of strings simply is a sequence of transitions that sucessfully maps the machine into some state, consuming some part of the input string and some part of the outpu string. We do not require that the partial run is part of a successful run, otherwise we would require to know what we are about to find out: whether any of the partial runs can be completed. We start with a single partial run: the empty run — no action taken. In the first step we list all transitions that extend this by one step. There is only one, $t(0)$. So, the next set of runs is $t(0)$. It leads to the situation $1, \mathsf{a|aa} : \mathsf{b|bbb}$. From here we can do two things: $t(1)$ or $t(2)$. We do both, and note the result:

(169) Step 2

$t(0), t(1) : 1, \mathsf{aa|a} : \mathsf{b|bbb}$

$t(0), t(2) : 1, \mathsf{a|aa} : \mathsf{bb|bb}$

In the next round we try to extend the result by one more step. In each of the two, we can perform either $t(1)$ or $t(2)$:

(170) Step 3

$t(0), t(1), t(1) : 1, \mathsf{aaa|} : \mathsf{b|bbb}$

$t(0), t(1), t(2) : 1, \mathsf{aa|a} : \mathsf{bb|bb}$

$t(0), t(2), t(1) : 1, \mathsf{aa|a} : \mathsf{bb|bb}$

$t(0), t(2), t(2) : 1, \mathsf{a|aa} : \mathsf{bbb|b}$

In the fourth step, we have no choice in the first line: we have consumed the input, now we need to choose $t(2)$. In the other cases we have both options still.

(171) Step 4

$t(0), t(1), t(1), t(2) : 1, \mathtt{aaa}| : \mathtt{bb}|\mathtt{bb}$
$t(0), t(1), t(2), t(1) : 1, \mathtt{aaa}| : \mathtt{bb}|\mathtt{bb}$
$t(0), t(1), t(2), t(2) : 1, \mathtt{aaa}| : \mathtt{bbb}|\mathtt{b}$
$t(0), t(2), t(1), t(1) : 1, \mathtt{aaa}| : \mathtt{bbb}|\mathtt{b}$
$t(0), t(2), t(1), t(2) : 1, \mathtt{aa}|\mathtt{a} : \mathtt{bbb}|\mathtt{b}$
$t(0), t(2), t(2), t(1) : 1, \mathtt{aa}|\mathtt{a} : \mathtt{bbb}|\mathtt{b}$
$t(0), t(2), t(2), t(2) : 1, \mathtt{a}|\mathtt{aa} : \mathtt{bbbb}|$

And so on.

It should be noted that remembering each and every run is actually excessive. If two runs terminate in the same configuration (state plus pair of read strings), they can be extended in the same way. Basically, the number of runs initially shows exponential growth, while the number of configurations is quadratic in the length of the smaller of the strings. So, removing this excess can be vital for computational reaons. In Step 3 we had 4 runs and 3 different end situations, in Step 4 we have 7 runs, but only 4 different end configurations.

Thus we can improve the algorithm once more as follows. We do not keep a record of the run, only of its resulting configuration, which consists of the state and positions at the input and the output string. In each step we just calculate the possible next configurations for the situations that have recently been added. Each step will advance the positions by at least one, so we are sure to make progress. How long does this algorithm take to work? Let's count. The configurations are triples $\langle x, j, k\rangle$, where x is a state of the machine, i is the position of the character to the right of the bar on the input string, k the position of the character to the right of the bar on the output string. On $\langle \vec{x}, \vec{y}\rangle$ there are $|Q| \cdot |\vec{x}| \cdot |\vec{y}|$ many such triples. All the algorithm does is compute which ones are accessible from $\langle 0, 0, 0\rangle$. At the end, it looks whether there is one of the form $\langle q, |\vec{x}|\vec{y}|\rangle$, where $q \in F$ (q is accepting). Thus the algorithm takes $|Q| \cdot |\vec{x}| \cdot |\vec{y}|$ time, time propotional in both lengths. (Recall here the discussion of Section **??**. What we are computing is the reachability relation in the set of configuations. This can be done in linear time.)

One might be inclined to think that the first algorithm is faster because it goes into the run very quickly. However, it is possible to show that if the input is

appropriately chosen, it might take exponential time to reach an answer. So, on certain examples this algorithm is excessively slow. (Nevertheless, sometimes it might be much faster that the breadth first algorithm.)

Now we look ata different problem: given an input string, what are the possible output strings? We have already seen that there can be infinitely many, so one might need a regular expression to list them all. Let us not do that here. Let us look instead into ways of finding at least one output string. First, an artificial example. The automaton has the following transitions.

(172)
$$0 \xrightarrow{\mathtt{a}:\mathtt{c}} 1 \qquad 0 \xrightarrow{\mathtt{a}:\mathtt{d}} 2$$
$$1 \xrightarrow{\mathtt{a}:\mathtt{c}} 1 \qquad 2 \xrightarrow{\mathtt{a}:\mathtt{d}} 2$$
$$2 \xrightarrow{\mathtt{b}:\varepsilon} 3$$

0 is initial, and 1 and 3 are accepting. This machine has as input strings all strings of the form $\mathtt{a}^+\mathtt{b}?$. However, the relation it defines is this:

(173) $\{\langle \mathtt{a}^n, \mathtt{c}^n\rangle : n > 0\} \cup \{\langle \mathtt{a}^n\mathtt{b}, \mathtt{d}\rangle : n > 0\}$

The translation of each given string is unique; however, it depends on the last character of the input! So, we cannot simply think of the machine as taking each character from the input and immediately returning the corresponding output character. We have to buffer the entire output until we are sure that the run we are looking at will actually succeed.

Let us look at a real problem. Suppose we design a transducer for Arabic. Let us say that the surface form `katab` should be translated into the deep form `ktb+3Pf`, but the form `baktib` into `ktb+1Pr`. Given: input string `baktib`. Configuration is now: 0, |`baktib`. No output yet. The first character is `b`. We face two choices: consider it as part of a root (say, of a form `badan` to be translated as `bdn+3Pf`), or to consider it is the first letter of the prefix `ba`. So far there is nothing that we can do to eliminate either option, so we keep them both open. It is only when we have reached the fourth letter, `t`, when the situation is clear: if this was a 3rd perfect form, we should see `a`. Thus, the algorithm we propose is this: try to find a run that matches the input string and then look at the output string it defines. The algorithm to find a run for the input string can be done as above, using depth first or breadth first search. Once again, this can be done in quadratic time. (Basically, the possible runs can be compactly represented and then converted into an output string.)

18 Context Free Grammars

Definition 22 *Let A be as usual an alphabet. Let N be a set of symbols, N disjoint with A, $\Sigma \subseteq N$. A **context free rule** over N as the set of **nonterminal symbols** and A the set of **terminal symbols** is a pair $\langle X, \vec{\alpha} \rangle$, where $X \in N$ and $\vec{\alpha}$ is a string over $N \cup A$. We write $X \to \vec{\alpha}$ for the rule. A **context free grammar** is a quadruple $\langle A, N, \Sigma, R \rangle$, where R is a finite set of context free rules. The set Σ is called the set of **start symbols***

Often it is assumed that a context free grammar has just one start symbol, but in actual practice this is mostly not observed. Notice that context free rules allow the use of nonterminals and terminals alike. The implicit convention we use is as follows. A string of terminals is denoted by a Roman letter with a vector arrow, a mixed string containing both terminals and nonterminals is denoted by a Greek letter with a vector arrow. (The vector is generally used to denote strings.) If the difference is not important, we shall use Roman letters.

The following are examples of context free rules. The conventions that apply here are as follows. Symbols in print are terminal symbols; they are denoted using typewriter font. Nonterminal symbols are produced by enclosing an arbitrary string in brackets: $< \cdots >$.

(174) $< \text{digit} > \to \texttt{0}$
$< \text{digit} > \to \texttt{1}$
$< \text{number} > \to < \text{digit} >$
$< \text{number} > \to < \text{number} >< \text{digit} >$

There are some conventions that apply to display the rules in a more compact form. The vertical slash '|' is used to merge two rules with the same left hand symbol. So, when $X \to \vec{\alpha}$ and $X \to \vec{\gamma}$ are rules, you can write $X \to \vec{\alpha} \mid \vec{\gamma}$. Notice that one speaks of one rule in the latter case, but technically we have two rules now. This allows us to write as follows.

(175) $< \text{digit} > \to \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \mid \texttt{9}$

And this stands for ten rules. Concatenation is not written. It is understood that symbols that follow each other are concatenated the way they are written down.

A context free grammar can be seen as a statement about sentential structure, or as a device to generate sentences. We begin with the second viewpoint. We write $\vec{\alpha} \Rightarrow \vec{\gamma}$ and say that $\vec{\gamma}$ is **derived** from $\vec{\alpha}$ **in one step** if there is a rule $X \to \vec{\eta}$, a single occurrence of X in $\vec{\alpha}$ such that $\vec{\gamma}$ is obtained by replacing that occurrence of X in $\vec{\alpha}$ by $\vec{\eta}$:

(176) $\vec{\alpha} = \vec{\sigma}^\frown X^\frown \vec{\tau}, \qquad \vec{\gamma} = \vec{\sigma}^\frown \vec{\eta}^\frown \vec{\tau}$

For example, if the rule is < number >→< number >< digit > then replacing the occurrence of < number > in the string 1 < digit >< number > 0 will give 1 < digit >< number >< digit > 0. Now write $\vec{\alpha} \Rightarrow^{n+1} \vec{\gamma}$ if there is a $\vec{\rho}$ such that $\vec{\alpha} \Rightarrow^n \vec{\rho}$ and $\vec{\rho} \Rightarrow \vec{\gamma}$. We say that $\vec{\gamma}$ is **derivable from** $\vec{\alpha}$ **in** $n + 1$ **steps**. Using the above grammar we have:

(177) < number >⇒^{4}< digit >< digit >< digit >< number >

Finally, $\vec{\alpha} \Rightarrow^* \vec{\gamma}$ ($\vec{\gamma}$ **is derivable from** $\vec{\alpha}$) if there is an n such that $\vec{\alpha} \Rightarrow^n \vec{\gamma}$. Now, if $X \Rightarrow^* \vec{\gamma}$ we say that $\vec{\gamma}$ is a **string of category** X. The **language** generated by G consists of all *terminal* strings that have category $S \in \Sigma$. Equivalently,

(178) $L(G) := \{\vec{x} \in A^* : \text{there is } S \in \Sigma : S \Rightarrow^* \vec{x}\}$

You may verify that the grammar above generates all strings of digits from the symbol < number >. If you take this as the start symbol, the language will consist in all strings of digits. If, however, you take the symbol < digit > as the start symbol then the language is just the set of all strings of length one. This is because even though the symbol < number > is present in the nonterminals and the rules, there is no way to generate it by applying the rules in succession from < digit >. If on the other hand, Σ contains both symbols then you get just the set of all strings, since a digit also is a number. (A fourth possibility is $\Sigma = \varnothing$, in which case the language is empty.)

There is a trick to reduce the set of start symbols to just one. Introduce a new symbol S^* together with the rules

(179) $S^* \to S$

for every $S \in \Sigma$. This is a different grammar but it derives the same strings. Notice that actual linguistics is different. In natural language having a set of start

symbols is very useful. There are several basic types of sentences (assertions, questions, commands, and so on). These represent different sets of strings. It real grammars (GPSG, HPSG, for example), one does not start from a single symbol but rather from different symbols, one for each class of saturated utterance. This is useful also for other reasons. It is noted that in answers to questions almost any constituent can function as a complete utterance. To account for that, one would need to enter a rule of the kind above for each constituent. But there is a sense in which these constituents are not proper sentences, they are just parts of constituents. Rather than having to write new rules to encompass the use of these constituents, we can just place the burden on the start set. So, in certain circumstances what shifts is the set of start symbols, the rule set however remains constant.

A **derivation** is a full specification of the way in which a string has been generated. This may be given as follows. It is a sequence of strings, where each subsequent string is obtained by applying a rule to some occurrence of a nonterminal symbol. We need to specify in each step which occurrence of a nonterminal is targeted. We can do so by underlining it. For example, here is a grammar.

(180) $\mathtt{A} \to \mathtt{BA} \mid \mathtt{AA}, \mathtt{B} \to \mathtt{AB} \mid \mathtt{BB}, \mathtt{A} \to \mathtt{a}, \mathtt{B} \to \mathtt{b} \mid \mathtt{bc} \mid \mathtt{cb}$

Here is a derivation:

(181) $\underline{\mathtt{A}}$, $\mathtt{B}\underline{\mathtt{A}}$, $\mathtt{BB}\underline{\mathtt{A}}$, $\mathtt{BBB}\underline{\mathtt{A}}$, $\mathtt{B}\underline{\mathtt{B}}\mathtt{Ba}$, $\underline{\mathtt{B}}\mathtt{cbBa}$, $\mathtt{bcb}\underline{\mathtt{B}}\mathtt{a}$, bcbbca

Notice that without underlining the symbol which is being replaced some information concerning the derivation is lost. In the second step we could have applied the rule $\mathtt{A} \to \mathtt{BA}$ instead, yielding the same result. Thus, the following also is a derivation:

(182) $\underline{\mathtt{A}}$, $\mathtt{B}\underline{\mathtt{A}}$, $\mathtt{BB}\underline{\mathtt{A}}$, $\mathtt{BBB}\underline{\mathtt{A}}$, $\mathtt{B}\underline{\mathtt{B}}\mathtt{Ba}$, $\underline{\mathtt{B}}\mathtt{cbBa}$, $\mathtt{bcb}\underline{\mathtt{B}}\mathtt{a}$, bcbbca

(I leave it to you to figure out how one can identify the rule that has been applied.) This type of information is essential, however. To see this, let us talk about the second approach to CFGs, the analytic one. Inductively, on the basis of a derivation, we assign constituents to the strings. First, however, we need to fix some notation.

Definition 23 *Let $\vec{x}$ and $\vec{y}$ be strings. An **occurrence** of $\vec{y}$ in $\vec{x}$ is a pair $\langle \vec{u}, \vec{v} \rangle$ of strings such that $\vec{x} = \vec{u}\vec{y}\vec{v}$. We call $\vec{u}$ the **left context of** $\vec{y}$ **in** $\vec{x}$ and $\vec{v}$ the **right context**.*

For example, the string aa has three occurrences in aacaaa: $\langle \varepsilon, \mathtt{caaa} \rangle$, $\langle \mathtt{aac}, \mathtt{a} \rangle$ and $\langle \mathtt{aaca}, \varepsilon \rangle$. It is *very* important to distinguish a substring from its occurrences in a string. Often we denote a particular occurrence of a substring by underlining it, or by other means. In parsing it is very popular to assign numbers to the positions between (!) the letters. Every letter then spans two adjacent positions, and in general a substring is represented by a pair of positions $\langle i, j \rangle$ where $i \leq j$. (If $i = j$ we have on occurrence of the empty string.) Here is an example:

(183)
E d d y
0 1 2 3 4

The pair $\langle 2, 3 \rangle$ represents the occurrence $\langle \mathtt{Ed}, \mathtt{y} \rangle$ of the letter d. The pair $\langle 1, 2 \rangle$ represents the occurrence $\langle \mathtt{E}, \mathtt{dy} \rangle$. The pair $\langle 1, 4 \rangle$ represents the (unique) occurrence of the string ddy. And so on. Notice that the strings are representatives of the strings, and they can represent the substring only in virtue of the fact that the larger string is actually given. This way of talking recommends itself when the larger string is fixed and we are only talking about its substrings (which is the case in parsing). For present purposes, it is however less explicit.

Let us return to our derivation. The derivation may start with any string $\vec{\alpha}$, but it is useful to think of the derivation as having started with a start symbol. Now, let the final string be $\vec{\gamma}$ (again, it need not be a terminal string, but it is useful to think that it is). We look at derivation (**??**). The last string is bcbbca. The last step is the replacement of B in bcbBa by bc. It is this replacement that means that the occurrence $\langle \mathtt{bcb}, \mathtt{a} \rangle$ of the string bc is a constituent of category B.

(184) bcb<u>bc</u>a

The step before that was the replacement of the first B by b. Thus, the first occurrence of b is a constituent of category b. The third last step was the replacement of the second B by cb, showing us that the occurrence $\langle \mathtt{b}, \mathtt{bca} \rangle$ is a constituent of category B. Then comes the replacement of A by a. So, we have the following first level constituents: $\langle \varepsilon, \mathtt{b} \rangle$, $\langle \mathtt{b}, \mathtt{bca} \rangle$, $\langle \mathtt{bcb}, \mathtt{a} \rangle$ and $\langle \mathtt{bcbbc}, \varepsilon \rangle$. Now we get to the replacement of B by BB. Now, the occurrence of BB in the string corresponds to the sequence of the occurrences $\langle \mathtt{b}, \mathtt{bca} \rangle$ of cb and $\langle \mathtt{bcb}, \mathtt{a} \rangle$ of bc. Their concatenation is cbbc, and the occurrence is $\langle \mathtt{b}, \mathtt{a} \rangle$.

(185) b<u>cbbc</u>a

We go one step back and find that, since now the first B is replaced by BB, the new constituent that we get is

(186) <u>bcbbc</u>a

which is of category B. In the last step we get that the entire string is of category A under this derivation. Thus, each step in a derivation adds a constituent to the analysis of the final string. The structure that we get is as follows. There is a string $\vec{\alpha}$ and a set Γ of occurrences of substrings of $\vec{\alpha}$ together with a map f that assigns to each member of Γ a nonterminal (that is, an element of N). We call the members of Γ **constituents** and $f(C)$ the **category** of C, $C \in \Gamma$.

By comparison, the derivation (**??**) imposes a different constituent structure on the string. The different is that it is not (**??**) that is a constituent but rather

(187) bcbbca

It is however not true that the constituents identify the derivation uniquely. For example, the following is a derivation that gives the same constituents as (**??**).

(188) A, BA, BBA, BBBA, BBBa, BBbca, bBbca, bcbbca

The difference between (**??**) and (**??**) are regarded inessential, however. Basically, only the constituent structure is really important, because it may give rise to different meanings, while different derivations that yield the same structure will give the same meaning.

The constituent structure is displayed by means of **trees**. Recall the definition of a tree. It is a pair $\langle T, < \rangle$ where $<$ is transitive (that is, if $x < y$ and $y < z$ then also $x < z$), it has a root (that is, there is a x such that for all $y \neq x$: $y < x$) and furthermore, if $x < y, z$ then either $y < z$, $y = z$ or $y > z$. We say that x **dominates** y if $x > y$; and that it **immediately dominates** y if it dominates y but there is no z such that x dominates z and z dominates y.

Now, let us return to the constituent structure. Let $C = \langle \vec{\gamma}_1, \vec{\gamma}_2 \rangle$ and $D = \langle \vec{\eta}_1, \vec{\eta}_2 \rangle$ be occurrences of substrings. We say that C is **dominated by** D, in symbols $C \prec D$, if $C \neq D$ and (1) $\vec{\eta}_1$ is a prefix of $\vec{\gamma}_1$ and (2) $\vec{\eta}_2$ is a suffix of $\vec{\gamma}2$. (It may happen that $\vec{\eta}_1 = \vec{\gamma}_1$ or that $\vec{\eta}_2 = \vec{\gamma}_2$, but not both.) Visually, what the definition amounts to is that if one underlines the substring of C and the substring of D then the latter line includes everything that the former underlines. For example, let $C = \langle \mathtt{abc}, \mathtt{dx} \rangle$ and $\langle \mathtt{a}, \mathtt{x} \rangle$. Then $C \prec D$, as they are different and (1) and (2) are satisfied. Visually this is what we get.

(189) abccddx

Now, let the tree be defined as follows. The set of nodes is the set Γ. The relation $<$ is defined by $\prec$.

The trees used in linguistic analysis are often ordered. The ordering is here implicitly represented in the string. Let $C = \langle \vec{\gamma}_1, \vec{\gamma}_2 \rangle$ be an occurrences of $\vec{\sigma}$ and $D = \langle \vec{\eta}_1, \vec{\eta}_2 \rangle$ be an occurrence of τ. We write $C \sqsubset D$ and say that C precedes D if $\vec{\gamma}_1\vec{\sigma}$ is a prefix of $\vec{\eta}_1$ (the prefix need not be proper). If one underlines C and D this definition amounts to saying that the line of C ends before the line of D starts.

(190) abccddx

Here $C = \langle \mathtt{a}, \mathtt{cddx} \rangle$ and $D = \langle \mathtt{abcc}, \mathtt{x} \rangle$.

19 Parsing and Recognition

Given a grammar G, and a string $\vec{x}$, we ask the following questions:

- (Recognition:) Is $\vec{x} \in L(G)$?
- (Parsing:) What derivation(s) does $\vec{x}$ have?

Obviously, as the derivations give information about the meaning associated with an expression, the problem of recognition is generally not of interest. Still, sometimes it is useful to first solve the recognition task, and then the parsing task. For if the string is not in the language it is unnecessary to look for derivations. The parsing problem for context free languages is actually not the one we are interested in: what we really want is only to know which constituent structures are associated with a given string. This vastly reduces the problem, but still the remaining problem may be very complex. Let us see how.

Now, in general a given string can have any number of derivations, even infinitely many. Consider by way of example the grammar

(191) $\mathtt{A} \to \mathtt{A} \mid \mathtt{a}$

It can be shown that if the grammar has no unary rules and nor rules of the form $X \to \varepsilon$ then a given string $\vec{x}$ has an exponential number of derivations. We shall show that it is possible to eliminate these rules (this reduction is not semantically innocent!). Given a rule $\mathtt{X} \to \varepsilon$ and a rule that contains X on the right, say

`Z` $\rightarrow$ `UXVX`, we eliminate the first rule (`X` $\rightarrow \varepsilon$); furthermore, we add all rules obtained by replacing any number of occurrences of `X` on the right by the empty string. Thus, we add the rules `Z` $\rightarrow$ `UVX`, `Z` $\rightarrow$ `UXV` and `Z` $\rightarrow$ `UV`. (Since other rules may have `X` on the left, it is not advisable to replace all occurrences of `X` uniformly.) We do this for all such rules. The resulting grammar generates the same set of strings, with the same set of constituents, excluding occurrences of the empty string. Now we are still left with unary rules, for example, the rule `X` $\rightarrow$ `Y`. Let ρ be a rule having `Y` on the left. We add the rule obtained by replacing `Y` on the left by `X`. For example, let `Y` $\rightarrow$ `UVX` be a rule. Then we add the rule `X` $\rightarrow$ `UVX`. We do this for all rules of the grammar. Then we remove `X` $\rightarrow$ `Y`.

These two steps remove the rules that do not expand the length of a string. We can express this formally as follows. If $\rho = X \rightarrow \vec{\alpha}$ is a rule, we call $|\vec{\alpha}| - 1$ the **productivity** of ρ, and denote it by $p(\rho)$. Clearly, $p(\rho) \geq -1$. If $p(\rho) = -1$ then $\vec{\alpha} = \varepsilon$, and if $p(\rho) = 0$ then we have a rule of the form $X \rightarrow Y$. In all other cases, $p(\rho) > 0$ and we call ρ **productive**.

Now, if $\vec{\eta}$ is obtained in one step from $\vec{\gamma}$ by use of ρ, then $|\vec{\eta}| = |\vec{\gamma}| + p(\rho)$. Hence $|\vec{\eta}| > |\vec{\gamma}|$ if $p(\rho) > 0$, that is, if ρ is productive. So, if the grammar only contains productive rules, each step in a derivation increases the length of the string, unless it replaces a nonterminal by a terminal. It follows that a string of length n has derivations of length $2n-1$ at most. Here is now a very simple minded strategy to find out whether a string is in the language of the grammar (and to find a derivation if it is): let $\vec{x}$ be given, of length n. Enumerate all derivations of length $< 2n$ and look at the last member of the derivation. If $\vec{x}$ is found once, it is in the language; otherwise not. It is not hard to see that this algorithm is exponential. We shall see later that there are far better algorithms, which are polynomial of order 3. Before we do so, let us note, however, that there are strings which have exponentially many different constituents, so that the task of enumerating the derivations is exponential. However, it still is the case that we can represent them is a very concise way, and this again takes only exponential time.

The idea to the algorithm is surprisingly simple. Start with the string $\vec{x}$. Scan the string for a substring $\vec{y}$ which occurs to the right of a rule $\rho = X \rightarrow \vec{y}$. Then write down all occurrences $C = \langle \vec{u}, \vec{v} \rangle$ (which we now represent by pairs of positions — see above) of $\vec{y}$ and declare them constituents of category X. There is an

actual derivation that defines this constituent structure:

(192) $\vec{u}\underline{X}\vec{v}, \vec{u}\vec{y}\vec{v} = \vec{x}$

We scan the string for each rule of the grammar. In doing so we have all possible constituents for derivations of length 1. Now we can discard the original string, and work instead with the strings obtained by undoing the last step. In the above case we analyze $\vec{u}X\vec{v}$ in the same way as we did with $\vec{x}$.

In actual practice, there is a faster way of doing this. All we want to know is what substrings qualify as constituents of some sort. The entire string is in the language if it qualifies as a constituent of category S for some $S \in \Sigma$. The procedure of establishing the categories is as follows. Let $\vec{x}$ be given, length n. Constituents are represented by pairs $[i, \delta]$, where i is the first position and $i+\delta$ the last. (Hence $0 < \delta \leq n$.) We define a matrix M of dimension $(n+1)\times n$. The entry $m(i, j)$ is the set of categories that the constituent $[i, j]$ has given the grammar G. (It is clear that we do not need to fill the entries $m(i, j)$ where $i + j > n$. They simply remain undefined or empty, whichever is more appropriate.) The matrix is filled inductively, starting with $j = 1$. We put into $m(i, 1)$ all symbols X such that $X \to x$ is a rule of the grammar, and x is the string between i and $i + 1$. Now assume that we have filled $m(i, k)$ for all $k < j$. Now we fill $m(i, j)$ as follows. For every rule $X \to \vec{\alpha}$ check to see whether the string between the positions i and $i + k$ has a decomposition as given by $\vec{\alpha}$. This can be done by cutting the string into parts and checking whether they have been assigned appropriate categories. For example, assume that we have a rule of the form

(193) $\mathtt{X} \to \mathtt{AbuXV}$

Then $\vec{x} = [i, j]$ is a string of category X if there are numbers k, m, n, p such that $[i, k]$ is of category $\mathtt{A}$, $[i + k, m] = \mathtt{bu}$ (so $m = 2$), $[i + k + m, n]$ is of category $\mathtt{X}$ and $[i + k + m + n, p]$ is of category $\mathtt{V}$, and, finally $k + m + n + p = k$. This involves choosing three numbers, k, n and p, such that $k + 2 + n + p = j$, and checking whether the entry $m(i, k)$ contains $\mathtt{A}$, whether $m(i+k+2, n)$ contains $\mathtt{X}$ and whether $m(i+k+2+n+p)$ contains $\mathtt{V}$. The latter entries have been computed, so this is just a matter of looking them up. Now, given k and n, p is fixed since $p = j-k-2-n$. There are $O(k^2)$ ways to choose these numbers. When we have filled the relevant entries of the matrix, we look up the entry $m(0, n)$. If it contains a $S \in \Sigma$ the string is in the language. (Do you see why?)

The algorithm just given is already polynomial. To see why, notice that in each step we need to cut up a string into a given number of parts. Depending on the

number ν of nonterminals to the right, this takes $O(n^{\nu-1})$ steps. There are $O(n^2)$ many steps. Thus, in total we have $O(n^{\nu+1})$ many steps to compute, each of which consumes time proportional to the size of the grammar. The best we can hope for is to have $\nu = 2$ in which case this algorithm needs time $O(n^3)$. In fact, this can always be achieved. Here is how. Replace the rule $\mathtt{X} \to \mathtt{AbuXV}$ be the following set of rules.

(194) $\mathtt{X} \to \mathtt{AbuY}, \mathtt{Y} \to \mathtt{XV}.$

Here, Y is assumed not to occur in the set of nonterminals. We remark without proof that the new grammar generates the same language. In general, a rule of the form $X \to Y_1 Y_2 \dots Y_n$ is replaced by

(195) $X \to Y_1 Z_1, Z_1 \to Y_2 Z_2, \dots, Z_{n-2} \to Y_{n-1} Y_n$

So, given this we can recognize the language in $O(n^3)$ time!

Now, given this algorithm, can we actually find a derivation or a constituent structure for the string in question? This can be done: start with $m(0, n)$. It contains a $S \in \Sigma$. Pick one of them. Now choose i such that $0 < i < n$ and look up the entries $m(0, i)$ and $m(i, n-i)$. If they contain A and B, respectively, and if $S \to AB$ is a rule, then begin the derivation as follows:

(196) $\underline{S}, AB$

Now underline one of A or B and continue with them in the same way. This is a downward search in the matrix. Each step requires linear time, since we only have to choose a point to split up the constituent. The derivation is linear in the length of the string. Hence the overall time is quadratic! Thus, surprisingly, the recognition consumes most of the work. Once that is done, parsing is easy.

Notice that the grammar transformation adds new constituents. In the case of the rule above we have added a new nonterminal Y and added the rules (**??**). However, it is easy to return to the original grammar: just remove all constituents of category Y. It is perhaps worth examining why adding the constituents saves time in parsing. The reason is simply that the task of identifying constituents occurs over and over again. The fact that a sequence of two constituents has been identified is knowledge that can save time later on when we have to find such a sequence. But in the original grammar there is no way of remembering that we did have it. Instead, the new grammar provides us with a constituent to handle the sequence.

Definition 24 *A CFG is in **standard form** if all the rules have the form* $X \to Y_1 Y_2 \cdots Y_n$*, with* $X, Y_i \in N$*, or* $X \to \vec{x}$*. If in addition* $n = 2$ *for all rules of the first form, the grammar is said to be in **Chomsky Normal Form**.*

There is an easy way to convert a grammar into standard form. Just introduce a new nonterminal Y_a for each letter $a \in A$ together with the rules $Y_a \to a$. Next replace each terminal a that cooccurs with a nonterminal on the right hand side of a rule by Y_a. The new grammar generates more constituents, since letters that are introduced together with nonterminals do not form a constituent of their own in the old grammar. Such letter occurrences are called **syncategorematic**. Typical examples of syncategorematic occurrences of letters are brackets that are inserted in the formation of a term. Consider the following expansion of the grammar (**??**).

(197) $<$ term $>\to$ $<$ number $>|$ (<term>+<term>)
$|$ (<term>*<term>)

Here, operator symbols as well as brackets are added syncategorematically. The procedure of elimination will yield the following grammar.

(198) $<$ term $>\to$ $<$ number $>| Y_(<$ term $> Y_+ <$ term $> Y_)$
$| Y_(<$ term $> Y_+ <$ term $> Y_)$
$Y_(\to$(
$Y_) \to$)
$Y_+ \to$+
$Y_* \to$*

However, often the conversion to standard form can be avoided however. It is mainly interesting for theoretic purposes.

Now, it may happen that a grammar uses more nonterminals than necessary. For example, the above grammar distinguishes Y_+ from Y_*, but this is not necessary. Instead the following grammar will just as well.

(199) $<$ term $>\to$ $<$ number $>| Y_(<$ term $> Y_o <$ term $> Y_)$
$Y_(\to$(
$Y_) \to$)
$Y_o \to$+ | *

The reduction of the number of nonterminals has the same significance as in the case of finite state automata: it speeds up recognition, and this can be significant because not only the number of states is reduced but also the number of rules.

Another source of time efficiency is the number of rules that match a given right hand side. If there are several rules, we need to add the left hand symbol for each of them.

Definition 25 *A CFG is **invertible** if for any pair of rules $X \to \vec{\alpha}$ and $Y \to \vec{\alpha}$ we have $X = Y$.*

There is a way to convert a given grammar into invertible form. The set of nonterminals is $\wp(N)$, the powerset of the set of nonterminals of the original grammar. The rules are

$$S \to T_0 T_1 \dots T_{n-1} \tag{200}$$

where S is the set of all X such that there are $Y_i \in T_i$ $(i < n)$ such that $X \to Y_0 Y_1 \dots Y_{n-1} \in R$. This grammar is clearly invertible: for any given sequence $T_0 T_1 \cdots T_{n-1}$ of nonterminals the left hand side S is uniquely defined. What needs to be shown is that it generates the same language (in fact, it generates the same constituent structures, though with different labels for the constituents).

20 Greibach Normal Form

We have spoken earlier about different derivations defining the same constituent structure. Basically, if in a given string we have several occurrences of nonterminals, we can choose any of them and expand them first using a rule. This is because the application of two rules that target the same string but different nonterminals commute:

$$\begin{array}{ccc} & \cdots X \cdots Y \cdots & \\ \cdots \vec{\alpha} \cdots Y \cdots & & \cdots X \cdots \vec{\gamma} \cdots \\ & \cdots \vec{\alpha} \cdots \vec{\gamma} \cdots & \end{array} \tag{201}$$

This can be exploited in many ways, for example by always choosing a particular derivation. For example, we can agree to always expand the leftmost nonterminal, or always the rightmost nonterminal.

Recall that a derivation defines a set of constituent occurrences, which in turn constitute the nodes of the tree. Notice that each occurrence of a nonterminal is replaced by some right hand side of a rule during a derivation that leads to a terminal string. After it has been replaced, it is gone and can no longer figure in a derivation. Given a tree, a **linearization** is an ordering of the nodes which results from a valid derivation in the following way. We write $x \lhd y$ if the constituent of x is expanded before the constituent of y is. One can characterize exactly what it takes for such an order to be a linearization. First, it is linear. Second if $x > y$ then also $x \lhd y$. It follows that the root is the first node in the linearization.

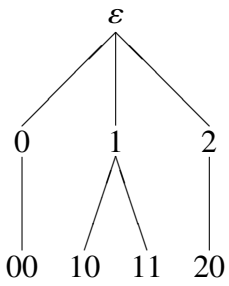
Linearizations are closely connected with search strategies in a tree. We shall present examples. The first is a particular case of the so–called **depth–first** search and the linearization shall be called **leftmost linearization**. It is as follows. $x \lhd y$ iff $x > y$ or $x \sqsubset y$. (Recall that $x \sqsubset y$ iff x precedes y. Trees are always considered ordered.) For every tree there is exactly one leftmost linearization. We shall denote the fact that there is a leftmost derivation of $\vec{\alpha}$ from X by $X \vdash_G^{\ell} \vec{\alpha}$. We can generalize the situation as follows. Let $\blacktriangleleft$ be a linear ordering uniformly defined on the leaves of local subtrees. That is to say, if $\mathfrak{B}$ and $\mathfrak{C}$ are isomorphic local trees (that is, if they correspond to the same rule ρ) then $\blacktriangleleft$ orders the leaves $\mathfrak{B}$ linearly in the same way as $\lhd$ orders the leaves of $\mathfrak{C}$ (modulo the unique (!) isomorphism). In the case of the leftmost linearization the ordering is the one given by $\sqsubset$. Now a minute's reflection reveals that every linearization of the local subtrees of a tree induces a linearization of the entire tree but not conversely (there are orderings which do not proceed in this way, as we shall see shortly). $X \vdash_G^{\blacktriangleleft} \vec{\alpha}$ denotes the fact that there is a derivation of $\vec{\alpha}$ from X determined by $\blacktriangleleft$. Now call π a priorization for $G = \langle \mathrm{S}, N, A, R\rangle$ if π defines a linearization on the local tree $\mathfrak{H}_\rho$, for every $\rho \in R$. Since the root is always the first element in a linearization, we only need to order the daughters of the root node, that is, the leaves. Let this ordering be $\blacktriangleleft$. We write $X \vdash_G^{\pi} \vec{\alpha}$ if $X \vdash_G^{\blacktriangleleft} \vec{\alpha}$ for the linearization $\blacktriangleleft$ defined by π.

Proposition 26 *Let π be a priorization. Then $X \vdash_G^{\pi} \vec{x}$ iff $X \vdash_G \vec{x}$.*

A different strategy is the *breadth–first search*. This search goes through the tree in increasing depth. Let S_n be the set of all nodes x with $d(x) = n$. For each n, S_n shall be ordered linearly by $\sqsubset$. The **breadth–first search** is a linearization Δ, which is defined as follows. (a) If $d(x) = d(y)$ then $x \Delta y$ iff $x \sqsubset y$, and (b) if $d(x) < d(y)$ then $x \Delta y$. The difference between these search strategies, depth–first and breadth–first, can be made very clear with tree domains.

Definition 27 *A **tree domain** is a set T of strings of natural numbers such that (i) if $\vec{x}$ if a prefix of $\vec{y} \in T$ then also $\vec{x} \in T$, (b) if $\vec{x}j \in T$ and $i < j$ then also $\vec{x}i \in T$. We define $\vec{x} > \vec{y}$ if $\vec{x}$ is a proper prefix of $\vec{y}$ and $\vec{x} \sqsubset \vec{y}$ iff $\vec{x} = \vec{u}i\vec{v}$ and $\vec{y} = \vec{u}j\vec{w}$ for some sequences $\vec{u}$, $\vec{v}$, $\vec{w}$ and numbers $i < j$.*

The depth–first search traverses the tree domain in the lexicographical order, the breadth–first search in the numerical order. Let the following tree domain be given.



The depth–first linearization is

(202) $\varepsilon, 0, 00, 1, 10, 11, 2, 20$

The breadth–first linearization, however, is

(203) $\varepsilon, 0, 1, 2, 00, 10, 11, 20$

Notice that with these linearizations the tree domain ω^* cannot be enumerated. Namely, the depth–first linearization begins as follows.

(204) $\varepsilon, 0, 00, 000, 0000, \ldots$

So we never reach 1. The breadth–first linearization goes like this.

(205) $\varepsilon, 0, 1, 2, 3, \ldots$

So, we never reach 00. On the other hand, ω^* is countable, so we do have a linearization, but it is more complicated than the given ones.

The first reduction of grammars we look at is the elimination of superfluous symbols and rules. Let $G = \langle \mathtt{S}, A, N, R\rangle$ be a CFG. Call $X \in N$ **reachable** if $G \vdash \vec{\alpha}^\frown X^\frown \vec{\beta}$ for some $\vec{\alpha}$ and $\vec{\beta}$. X is called **completable** if there is an $\vec{x}$ such that $X \Rightarrow_R^* \vec{x}$.

$$\begin{array}{llllll} & \mathtt{S} & \to & \mathtt{AB} & \mathtt{A} & \to & \mathtt{CB} \\ (206) & \mathtt{B} & \to & \mathtt{AB} & \mathtt{A} & \to & \mathtt{x} \\ & \mathtt{D} & \to & \mathtt{Ay} & \mathtt{C} & \to & \mathtt{y} \end{array}$$

In the given grammar A, C and D are completable, and S, A, B and C are reachable. Since S, the start symbol, is not completable, no symbol is both reachable and completable. The grammar generates no terminal strings.

Let N' be the set of symbols which are both reachable and completable. If $\mathtt{S} \notin N'$ then $L(G) = \emptyset$. In this case we put $N' := \{\mathtt{S}\}$ and $R' := \emptyset$. Otherwise, let R' be the restriction of R to the symbols from $A \cup N'$. This defines $G' = \langle \mathtt{S}, N', A, R'\rangle$. It may be that throwing away rules may make some nonterminals unreachable or uncompletable. Therefore, this process must be repeated until $G' = G$, in which case every element is both reachable and completable. Call the resulting grammar G^s. It is clear that $G \vdash \vec{\alpha}$ iff $G^s \vdash \vec{\alpha}$. Additionally, it can be shown that every derivation in G is a derivation in G^s and conversely.

Definition 28 *A CFG is called **slender** if either $L(G) = \emptyset$ and G has no nonterminals except for the start symbol and no rules; or $L(G) \neq \emptyset$ and every nonterminal is both reachable and completable.*

Two slender grammars have identical sets of derivations iff their rule sets are identical.

Proposition 29 *Let G and H be slender. Then $G = H$ iff* $\mathrm{der}(G) = \mathrm{der}(H)$.

Proposition 30 *For every CFG G there is an effectively constructible slender CFG $G^s = \langle \mathtt{S}, N^s, A, R^s\rangle$ such that $N^s \subseteq N$, which has the same set of derivations as G. In this case it also follows that $L_B(G^s) = L_B(G)$.* □

Definition 31 *Let $G = \langle \mathtt{S}, N, A, R\rangle$ be a CFG. G is in **Greibach (Normal) Form** if every rule is of the form $\mathtt{S} \to \varepsilon$ or of the form $X \to x^\frown \vec{Y}$.*

Proposition 32 *Let G be in Greibach Normal Form. If $X \vdash_G \vec{\alpha}$ then $\vec{\alpha}$ has a leftmost derivation from X in G iff $\vec{\alpha} = \vec{y}^\frown\vec{Y}$ for some $\vec{y} \in A^*$ and $\vec{Y} \in N^*$ and $\vec{y} = \varepsilon$ only if $\vec{Y} = X$.*

The proof is not hard. It is also not hard to see that this property characterizes the Greibach form uniquely. For if there is a rule of the form $X \to Y^\frown\vec{\gamma}$ then there is a leftmost derivation of $Y^\frown\vec{\gamma}$ from X, but not in the desired form. Here we assume that there are no rules of the form $X \to X$.

Theorem 33 (Greibach) *For every CFG one can effectively construct a grammar G^g in Greibach Normal Form with $L(G^g) = L(G)$.*

Before we start with the actual proof we shall prove some auxiliary statements. We call ρ an X–**production** if $\rho = X \to \vec{\alpha}$ for some $\vec{\alpha}$. Such a production is called **left recursive** if it has the form $X \to X^\frown\vec{\beta}$. Let $\rho = X \to \vec{\alpha}$ be a rule; define $R^{-\rho}$ as follows. For every factorization $\vec{\alpha} = \vec{\alpha}_1^\frown Y^\frown\vec{\alpha}_2$ of $\vec{\alpha}$ and every rule $Y \to \vec{\beta}$ add the rule $X \to \vec{\alpha}_1^\frown\vec{\beta}^\frown\vec{\alpha}_2$ to R and finally remove the rule ρ. Now let $G^{-\rho} := \langle \mathsf{S}, N, A, R^{-\rho}\rangle$. Then $L(G^{-\rho}) = L(G)$. We call this construction as **skipping** the rule ρ. The reader may convince himself that the tree for $G^{-\rho}$ can be obtained in a very simple way from trees for G simply by removing all nodes x which dominate a local tree corresponding to the rule ρ, that is to say, which are isomorphic to $\mathfrak{H}_\rho$. This technique works only if ρ is not an S–production. In this case we proceed as follows. Replace ρ by all rules of the form $\mathsf{S} \to \vec{\beta}$ where $\vec{\beta}$ derives from $\vec{\alpha}$ by applying a rule. Skipping a rule does not necessarily yield a new grammar. This is so if there are rules of the form $X \to Y$ (in particular rules like $X \to X$).

Lemma 34 *Let $G = \langle \mathsf{S}, N, A, R\rangle$ be a CFG and let $X \to X^\frown\vec{\alpha}_i$, $i < m$, be all left recursive X–productions as well as $X \to \vec{\beta}_j$, $j < n$, all non left recursive X–productions. Now let $G^1 := \langle \mathsf{S}, N \cup \{Z\}, A, R^1\rangle$, where $Z \notin N \cup A$ and R^1 consists of all Y–productions from R with $Y \neq X$ as well as the productions*

$$\begin{array}{llll} X \to \vec{\beta}_j & j < n, & Z \to \vec{\alpha}_i & i < m, \\ X \to \vec{\beta}_j^\frown Z & j < n, & Z \to \vec{\alpha}_i^\frown Z & i < m. \end{array} \tag{207}$$

Then $L(G^1) = L(G)$.

Proof. We shall prove this lemma rather extensively since the method is relatively tricky. We consider the following priorization on G^1. In all rules of the form $X \to \vec{\beta}_j$ and $Z \to \vec{\alpha}_i$ we take the natural ordering (that is, the leftmost ordering) and in all rules $X \to \vec{\beta}_j Z$ as well as $Z \to \vec{\alpha}_i Z$ we also put the left to right ordering except that Z precedes all elements from $\vec{\alpha}_j$ and $\vec{\beta}_i$, respectively. This defines the linearization $\blacktriangleleft$. Now, let $M(X)$ be the set of all $\vec{\gamma}$ such that there is a leftmost derivation from X in G in such a way that $\vec{\gamma}$ is the first element not of the form $X^\frown\vec{\delta}$. Likewise, we define $P(X)$ to be the set of all $\vec{\gamma}$ which can be derived from X priorized by $\blacktriangleleft$ in G^1 such that $\vec{\gamma}$ is the first element which does not contain Z. We claim that $P(X) = M(X)$. It can be seen that

$$M(X) = \bigcup_{j<n} \vec{\beta}_j \cdot (\bigcup_{i<m} \vec{\alpha}_i)^* = P(X) \tag{208}$$

From this the desired conclusion follows thus. Let $\vec{x} \in L(G)$. Then there exists a leftmost derivation $\Gamma = \langle A_i : i < n+1\rangle$ of $\vec{x}$. (Recall that the A_i are instances of rules.) This derivation is cut into segments Σ_i, $i < \sigma$, of length k_i, such that

$$\Sigma_i = \langle A_j : \sum_{p<i} k_p \leq j < 1 + \sum_{p<i+1} k_i\rangle \tag{209}$$

This partitioning is done in such a way that each Σ_i is a maximal portion of Γ of X–productions or a maximal portion of Y–productions with $Y \neq X$. The X–segments can be replaced by a $\blacktriangleleft$–derivation $\widehat{\Sigma_i}$ in G^1, by the previous considerations. The segments which do not contain X–productions are already G^1–derivations. For them we put $\widehat{\Sigma_i} := \Sigma_i$. Now let $\widehat{\Gamma}$ be result of stringing together the $\widehat{\Sigma_i}$. This is well–defined, since the first string of $\widehat{\Sigma_i}$ equals the first string of Σ_i, as the last string of $\widehat{\Sigma_i}$ equals the last string of Σ_i. $\widehat{\Gamma}$ is a G^1–derivation, priorized by $\blacktriangleleft$. Hence $\vec{x} \in L(G^1)$. The converse is analogously proved, by beginning with a derivation priorized by $\blacktriangleleft$. □

Now to the proof of Theorem **??**. We may assume at the outset that G is in Chomsky Normal Form. We choose an enumeration of N as $N = \{X_i : i < p\}$. We claim first that by taking in new nonterminals we can see to it that we get a grammar G^1 such that $L(G^1) = L(G)$ in which the X_i–productions have the form $X_i \to x^\frown\vec{Y}$ or $X_i \to X_j^\frown\vec{Y}$ with $j > i$. This we prove by induction on i. Let i_0 be the smallest i such that there is a rule $X_i \to X_j^\frown\vec{Y}$ with $j \leq i$. Let j_0 be the largest j such that $X_{i_0} \to X_j^\frown\vec{Y}$ is a rule. We distinguish two cases. The first is $j_0 = i_0$. By the previous lemma we can eliminate the production by introducing some new

nonterminal symbol Z_{i_0}. The second case is $j_0 < i_0$. Here we apply the induction hypothesis on j_0. We can skip the rule $X_{i_0} \to X_{j_0}\vec{Y}$ and introduce rules of the form (a) $X_{i_0} \to X_k\vec{Y}'$ with $k > j_0$. In this way the second case is either eliminated or reduced to the first.

Now let $P := \{Z_i : i < p\}$ be the set of newly introduced nonterminals. It may happen that for some j Z_j does not occur in the grammar, but this does not disturb the proof. Let finally $P_i := \{Z_j : j < i\}$. At the end of this reduction we have rules of the form

$$\text{(210a)} \quad X_i \to X_j\vec{Y} \qquad (j > i)$$

$$\text{(210b)} \quad X_i \to x\vec{Y} \qquad (x \in A)$$

$$\text{(210c)} \quad Z_i \to \vec{W} \qquad (\vec{W} \in (N \cup P_i)^+(\varepsilon \cup Z_i))$$

It is clear that every X_{p-1}–production already has the form $X_{p-1} \to x\vec{Y}$. If some X_{p-2}–production has the form (**??**) then we can skip this rule and get rules of the form $X_{p-2} \to \vec{x}\vec{Y}'$. Inductively we see that all rules of the form can be eliminated in favour of rules of the form (**??**). Now finally the rules of type (**??**). Also these rules can be skipped, and then we get rules of the form $Z \to x\vec{Y}$ for some $x \in A$, as desired.

For example, let the following grammar be given.

$$\text{(211)} \quad \begin{array}{lcllcl} \mathtt{S} & \to & \mathtt{SDA} \mid \mathtt{CC} & \mathtt{A} & \to & \mathtt{a} \\ \mathtt{D} & \to & \mathtt{DC} \mid \mathtt{AB} & \mathtt{B} & \to & \mathtt{b} \\ & & & \mathtt{C} & \to & \mathtt{c} \end{array}$$

The production $\mathtt{S} \to \mathtt{SDA}$ is left recursive. We replace it according to the above lemma by

$$\text{(212)} \quad \mathtt{S} \to \mathtt{CCZ}, \quad \mathtt{Z} \to \mathtt{DA}, \quad \mathtt{Z} \to \mathtt{DAZ}$$

Likewise we replace the production $\mathtt{D} \to \mathtt{DC}$ by

$$\text{(213)} \quad \mathtt{D} \to \mathtt{ABY}, \quad \mathtt{Y} \to \mathtt{C}, \quad \mathtt{Y} \to \mathtt{CY}$$

With this we get the grammar

$$\text{(214)} \quad \begin{array}{lcllcl} \mathtt{S} & \to & \mathtt{CC} \mid \mathtt{CCZ} & \mathtt{A} & \to & \mathtt{a} \\ \mathtt{Z} & \to & \mathtt{DA} \mid \mathtt{DAZ} & \mathtt{B} & \to & \mathtt{b} \\ \mathtt{D} & \to & \mathtt{AB} \mid \mathtt{ABY} & \mathtt{C} & \to & \mathtt{c} \\ \mathtt{Y} & \to & \mathtt{C} \mid \mathtt{CY} & & & \end{array}$$

Next we skip the D–productions.

$$
\begin{array}{lcllcl}
\mathtt{S} & \to & \mathtt{CC} \mid \mathtt{CCZ} & \mathtt{A} & \to & \mathtt{a} \\
\mathtt{Z} & \to & \mathtt{ABA} \mid \mathtt{ABYA} \mid \mathtt{ABAZ} \mid \mathtt{ABYAZ} & \mathtt{B} & \to & \mathtt{b} \\
\mathtt{D} & \to & \mathtt{AB} \mid \mathtt{ABY} & \mathtt{C} & \to & \mathtt{c} \\
\mathtt{Y} & \to & \mathtt{C} \mid \mathtt{CY} & & &
\end{array} \tag{215}
$$

Next D can be eliminated (since it is not reachable) and we can replace on the right hand side of the productions the first nonterminals by terminals.

$$
\begin{array}{lcl}
\mathtt{S} & \to & \mathtt{cC} \mid \mathtt{cCZ} \\
\mathtt{Z} & \to & \mathtt{aBA} \mid \mathtt{aBYA} \mid \mathtt{aBAZ} \mid \mathtt{aBYZ} \\
\mathtt{Y} & \to & \mathtt{c} \mid \mathtt{cY}
\end{array} \tag{216}
$$

Now the grammar is in Greibach Normal Form.

21 Pushdown Automata

Regular languages can be recognized by a special machine, the finite state automaton. Recognition here means that the machine scans the string left–to–right and when the string ends (the machine is told when the string ends) then it says 'yes' if the string is in the language and 'no' otherwise. (This picture is only accurate for deterministic machines; more on that later.)

There are context free languages which are not regular, for example $\{\mathtt{a}^n\mathtt{b}^n : n \in \mathbb{N}\}$. Thus devices that can check membership in $L(G)$ for a CFG must therefore be more powerful. The devices that can do this are called **pushdown automata**. They are finite state machines which have a memory in form of a pushdown. A pushdown memory is potentially infinite. You can store in it as much as you like. However, the operations that you can perform on a pushdown storage are limited. You can see only the last item you added to it, and you can either put something on top of that element, or you can remove that element and then the element that was added before it becomes visible. This behaviour is also called **LIFO** (last–in–first–out) storage. It is realized for example when storing plates. You always add plates on top, and remove from the top, so that the bottommost plates are only used when there is a lot of people. It is easy to see that you can decide whether a given string has the form $\mathtt{a}^n\mathtt{b}^n$ given an additional pushdown storage. Namely, you scan the string from left to right. As long as you get a, you put it on the

pushdown. Once you hit on a b you start popping the as from the pushdown, one for each b that you find. If the pushdown is emptied before the string is complete, then you have more bs than a. If the pushdown is not emptied but the string is complete, then you have more as than bs. So, you can tell whether you have a string of the required form if you can tell whether you have an empty storage. We assume that this is the case. In fact, typically what one does is to fill the storage before we start with a special symbol #, the **end-of-storage marker**. The storage is represented as a string over an alphabet D that contains #, the **storage alphabet**. Then we are only in need of the following operations and predicates:

❶ For each $d \in D$ we have an operation $\text{push}_d : \vec{x} \mapsto \vec{x}^\frown d$.

❷ For each $d \in D$ we have a predicate top_d which if true of $\vec{x}$ iff $\vec{x} = \vec{y}^\frown d$.

❸ We have an operation $\text{pop} : \vec{x}^\frown d \mapsto \vec{x}$.

(If we do not have an end of stack marker, we also need a predicate 'empty', which is true of a stack $\vec{x}$ iff $\vec{x} = \varepsilon$.

Now, notice that the control structure is a finite state automaton. It schedules the actions using the stack as a storage. This is done as follows. We have two alphabets, A, the alphabet of letters read from the tape, and I, the stack alphabet, which contains a special symbol, #. Initially, the stack contains one symbol, #. A transition instruction is a a quintuple $\langle s, c, t, s', p\rangle$, where s and s' are states, c is a character or empty (the character read from the string), and t is a character (read from the top of the stack) and finally p an instruction to either pop from the stack or push a character (different from #) onto it. A PDA contains a set of instructions. Formally, it is defined to be a quintuple $\langle A, I, Q, i_0, F, \sigma\rangle$, where A is the input alphabet, I the stack alphabet, Q the set of states, $i_0 \in Q$ the start state, $F \subseteq Q$ the set of accepting states, and σ a set of instructions. If $\mathfrak{A}$ is reading a string, then it does the following. It is initialized with the stack containing # and the initial state i_0. Each instruction is an option for the machine to proceed. However, it can use that option only if it is in state s, if the topmost stack symbol is t and if $c \neq \varepsilon$, the next character must match c (and is then consumed). The next state is s' and the stack is determined from p. If $p = \text{pop}$, then the topmost symbol is popped, if it is push_a, then a is pushed onto stack. PDAs can be nondeterministic. For a given situation we may have several options. If given the current stack, the current state and the next character there is at most one operation that can be chosen, we call

the PDA **deterministic**. We add here that theoretically the operation that reads the top of the stack removes the topmost symbol. The stack really is just a memory device. In order to look at the topmost symbol we actally need to pop it off the stack. However, if we put it back, then this as if we had just 'peeked' into the top of the stack. (We shall not go into the details here: but it is possible to peek into any number of topmost symbols. The price one pays is an exponential blowup of the number of states.)

We say that the PDA **accepts** $\vec{x}$ **by state** if it is in a final state at the end of $\vec{x}$. To continue the above example, we put the automaton in an accepting state if after popping a as the topmost symbol is #. Alternatively, we say that the PDA **accepts** $\vec{x}$ **by stack** if the stack is empty after $\vec{x}$ has been scanned. A slight modification of the machine results in a machine that accepts the language by stack. Basically, it needs to put one a less than needed on the stack and then cancel # on the last move. It can be shown that the class of languages accepted by PDAs by state is the same as the class of languages accepted by PDAs by stack, although for a given machine the two languages may be different. We shall establish that the class of languages accepted by PDAs by stack are exactly the CFGs. There is a slight problem in that the PDAs might actually be nondeterministic. While in the case of finite state automata there was a way to turn the machine into an equivalent deterministic machine, this is not possible here. There are languages which are CF but cannot be recognized by a deterministic PDA. An example is the language of palindromes: $\{\vec{x}\vec{x}^T : \vec{x} \in A^*\}$, where $\vec{x}^T$ is the reversal of $\vec{x}$. For example, $\mathtt{abddc}^T = \mathtt{cddba}$. The obvious mechanism is this: scan the input and start pushing the input onto the stack until you are half through the string, and then start comparing the stack with the string you have left. You accept the string if at the end the stack is #. Since the stack is popped in reverse order, you recognize exactly the palindromes. The trouble is that there is no way for the machine to know when to shift gear: it cannot tell when it is half through the string. Here is the dilemma. Let $\vec{x} = \mathtt{abc}$. Then abccba is a palindrome, but so is abccbaabccba and abccbaabccbaabccba. In general, $\mathtt{abccba}^n$ is a palindrome. If you are scanning a word like this, there is no way of knowing when you should turn and pop symbols, because the string might be longer than you have thought.

It is for this reason that we need to review our notion of acceptance. First, we say that a **run** of the machine is a series of actions that it takes, given the input. Alternatively, the run specifies what the machine chooses each time it faces a choice. (The alternatives are simply different actions and different subsequent states.) A

machine is deterministic iff for every input there is only one run. We say that the machine accepts the string if there is an accepting run on that input. Notice that the definition of a run is delicate. Computers are not parallel devices, they can only execute one thing at a time. They also are deterministic. The PDA has the same problem: it chooses a particular run but has no knowledge of what the outcome would have been had it chosen a different run. Thus, to check whether a run exists on a input we need to emulate the machine, and enumerate all possible runs and check the outcome. Alternatively, we keep a record of the run and backtrack.

Now back to the recognition problem. To show the theorem we use the Greibach Normal Form. Observe the following. The Greibach Form always puts a terminal symbol in front of a series of nonterminals. We define the following machine. Its stack alphabet is N, the beginning of stack is the start symbol of G. Now let $X \to aY_0Y_1 \cdots Y_{n-1}$ be a rule. Then we translate this rule into the following actions. Whenever the scanned symbol is a, and whenever the topmost symbol of the stack is X then pop that symbol from the stack and put Y_{n-1} then Y_{n-2} then Y_{n-3} etc. on the stack. (To schedule this correctly, the machine needs to go through several steps, since each step allows to put only one symbol onto the stack. But we ignore that finesse here.) Thus the last symbol put on the stack is Y_0, which is then visible. It can be shown that the machine ends with an empty stack on input $\vec{x}$ iff there is a leftmost derivation of $\vec{x}$ that corresponds to the run.

Let us see an example. Take the grammar

(217)
$$\begin{array}{llll} 0 & \mathtt{S} & \to & \mathtt{cC} \\ 1 & \mathtt{S} & \to & \mathtt{cCZ} \\ 2 & \mathtt{Z} & \to & \mathtt{aBA} \\ 3 & \mathtt{Z} & \to & \mathtt{aBYA} \\ 4 & \mathtt{Z} & \to & \mathtt{aBAZ} \\ 5 & \mathtt{Z} & \to & \mathtt{aBYZ} \\ 6 & \mathtt{Y} & \to & \mathtt{c} \\ 7 & \mathtt{Y} & \to & \mathtt{cY} \\ 8 & \mathtt{A} & \to & \mathtt{a} \\ 9 & \mathtt{B} & \to & \mathtt{b} \\ 10 & \mathtt{C} & \to & \mathtt{c} \end{array}$$

We take the string ccabccaba. Here is a leftmost derivation (to the left we show

the string, and to the right the last rule that we applied):

(218)

S	–
cCZ	1
ccZ	10
ccaBYZ	5
ccabYZ	9
ccabcYZ	7
ccabccZ	8
ccabccaBA	2
ccabccabA	9
ccabccaba	8

The PDA is parsing the string as follows. (We show the successful run.) The stack is initialized to S (Line 1). It reads c and deletes S, but puts first Z and then C on it. The stack now is ZC (leftmost symbol is at the bottom!). We are in Line 2. Now it reads c and deletes C from stack (Line 3). Then it reads a and pops Z, but pushes first Z, then Y and then B (Line 4). Then it pops B on reading b, pushing nothing on top (Line 5). It is clear that the strings to the left represent the following: the terminal string is that part of the input string that has been read, and the nonterminal string is the stack (in reverse order). As said, this is just one of the possible runs. There is an unsuccessful run which starts as follows. The stack is S. The machine reads c and decides to go with Rule 0: so it pops C but pushes only C. Then it reads c, and is forced to go with Rule 10, popping off C, but pushing nothing onto it. Now the stack is empty, and no further operation can take place. That run fails.

Thus, we have shown that context free languages can be recognized by PDAs by empty stack. The converse is a little trickier, we will not spell it out here.

22 Shift–Reduce–Parsing

We have seen above that by changing the grammar to Greibach Normal Form we can easily implement a PDA that recognizes the strings using a pushdown of the nonterminals. It is not necessary to switch to Greibach Normal Form, though. We can translate directly the grammar into a PDA. Alos, grammar and automaton are not uniquely linked to each other. Given a particular grammar, we can define quite

different automata that recognize the languages based on that grammar. The PDA implements what is often called a **parsing strategy**. The parsing strategy makes use of the rules of the grammar, but depending on the grammar in quite different ways. One very popular method of parsing is the following. We scan the string, putting the symbols one by one on the stack. Every time we hit a right hand side of a rule we undo the rule. This is to do the following: suppose the top of the stack contains the right hand side of a rule (in reverse order). Then we pop that sequence and push the left–hand side of the rule on the stack. So, if the rule is $\mathtt{A} \to \mathtt{a}$ and we get a, we first push it onto the stack. The top of the stack now matches the right hand side, we pop it again, and then push A. Suppose the top of the stack contains the sequence BA and there is a rule $\mathtt{S} \to \mathtt{AB}$, then we pop twice, removing that part and push S. To do this, we need to be able to remember the top of the stack. If a rule has two symbols to its right, then the top two symbols need to be remembered. We have seen earlier that this is possible, if the machine is allowed to do some empty transitions in between. Again, notice that the strategy is nondeterministic in general, because several options can be pursued at each step. (a) Suppose the top part of the stack is BA, and we have a rule $\mathtt{S} \to \mathtt{AB}$. Then either we push the next symbol onto the stack, or we use the rule that we have. (b) Suppose the top part of the stack is BA and we have the rules $\mathtt{S} \to \mathtt{AB}$ and $\mathtt{C} \to \mathtt{A}$. Then we may either use the first or the second rule.

The strongest variant is to always reduce when the right hand side of a rule is on top of the stack. Despite not being always successful, this strategy is actually useful in a number of cases. The condition under which it works can be spelled out as follows. Say that a CFG is **transparent** if for every string $\vec{\alpha}$ and nonterminal $X \Rightarrow^* \vec{\alpha}$, if there is an occurrence of a substring $\vec{\gamma}$, say $\vec{\alpha} = \vec{\kappa}_1\vec{\gamma}\vec{\kappa}_2$, and if there is a rule $Y \to \vec{\gamma}$, then $\langle\vec{\kappa}_1, \vec{\kappa}2\rangle$ is a constituent occurrence of category Y in $\vec{\alpha}$. This means that up to inessential reorderings there is just one parse for any given string if there is any. An example of a transparent language is arithmetical terms where no brackets are omitted. Polish Notation and Reverse Polish Notation also belong to this category. A broader class of languages where this strategy is successful is the class of NTS–languages. A grammar has the NTS–property if whenever there is a string $S \Rightarrow^* \vec{\kappa}_1\vec{\gamma}\vec{\kappa}_2$ and if $Y \to \vec{\gamma}$ then $S \Rightarrow^* \vec{\kappa}_1 Y\vec{\kappa}_2$. (Notice that this does not mean that the constituent is a constituent under the same parse; it says that one can find a parse that ends in Y being expanded in this way, but it might just be a different parse.) Here is how the concepts differ. There is a grammar for numbers that runs as follows.

(219) $< \text{digit} > \to \mathtt{0} \mid \mathtt{1} \mid \cdots \mid \mathtt{9}$

$$< \text{number} > \rightarrow \ < \texttt{digit} > | < \texttt{number} > < \texttt{number} >$$

This grammar has the NTS–property. Any digit is of category < digit >, and a number can be broken down into two numbers at any point. The standard definition is as follows.

(220)
$$< \text{digit} > \rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9}$$
$$< \text{number} > \rightarrow \ < \texttt{digit} > | < \text{digit} > < \text{number} >$$

This grammar is not an NTS–grammar, because in the expression

(221) $< \text{digt} > < \text{digit} > < \text{digit} >$

the occurrence $\langle < \texttt{digit} >, \varepsilon \rangle$ of the string < digit >< digit > cannot be a constituent occurrence under any parse. Finally, the language of strings has no transparent grammar! This is because the string `732` possesses occurrences of the strings `73` and `32`. These occurrences must be occurrences under one and the same parse if the grammar is transparent. But they overlap. Contradiction.

Now, the strategy above is deterministic. It is easy to see that there is a nondeterministic algorithm implementing this idea that actually defines a machine parsing exactly the CFLs. It is a machine where you only have a stack, and symbols scanned are moved onto the stack. The top k symbols are visible (with PDAs we had $k = 1$). If the last m symbols, $m \leq k$, are the right hand side of a rule, you are entitled to replace it by the corresponding left hand side. (This style of parsing is called shift–reduce parsing. One can show that PDAs can emulate a shift–reduce parser.) The stack gets cleared by $m - 1$ symbols, so you might end up seeing more of the stack after this move. Then you may either decide that once again there is a right hand side of a rule which you want to replace by the left hand side (reduce), or you want to see one more symbol of the string (shift). In general, like with PDAs, the nondeterminism is unavoidable.

There is an important class of languages, the so–called LR(k)–grammars. These are languages where the question whether to shift or to reduce can be based on a lookahead of k symbols. That is to say, the parser might not know directly if it needs to shift or to reduce, but it can know for sure if it sees the next k symbols (or whatever is left of the string). One can show that for a language which there is an LR(k)–grammar with $k > 0$ there also is a LR(1)–grammar. So, a lookahead of just one symbol is enough to make an informed guess (at least with respect to some grammar, which however need not generate the constituents we are interested in).

If the lookahead is 0 and we are interested in acceptance by stack, then the PDA also tells us when the string should be finished. Because in consuming the last letter it should know that this is the last letter because it will erase the symbol # at the last moment. Unlike nondeterministic PDAs which may have alternative paths they can follow, a deterministic PDA with out lookahead must make a firm guess: if the last letter is there it must know that this is so. This is an important class. Language of this form have the following property: no proper prefix of a string of the language is a string of the language. The language $B = \{\mathsf{a}^n\mathsf{b}^n : n \in \mathbb{N}\}$ of balanced strings is in LR(0), while for example B^+ is not (if contains the string `abab`, which has a proper prefix `ab`.)

23 Some Metatheorems

It is often useful to know whether a given language can be generated by geammars of a given type. For example: how do we decide whether a language is regular? The following is a very useful citerion.

Theorem 35 *Suppose that L is a regular language. Then there exists a number k such that every $\vec{x} \in L$ of length at least k has a decomposition $\vec{x} = \vec{u}\vec{v}\vec{w}$ with nonempty $\vec{v}$ such that for all $n \in \mathbb{N}$:*

$$\vec{u}\vec{v}^n\vec{w} \in L \tag{222}$$

Before we enter the proof, let us see some consequences of this theorem. First, we may choose $n = 0$, in which case we get $\vec{u}\vec{w} \in L$.

The proof is as follows. Since L is regular, there is a fsa $\mathfrak{A}$ such that $L = L(\mathfrak{A})$. Let k be the number of states of $\mathfrak{A}$. Let $\vec{x}$ be a string of length at least k. If $\vec{x} \in L$ then there is an accepting run of $\mathfrak{A}$ on $\vec{x}$:

$$q_0 \stackrel{x_0}{\to} q_1 \stackrel{x_1}{\to} q_2 \stackrel{x_2}{\to} q_3 \cdots q_{n-1} \stackrel{x_{n-1}}{\to} q_n \tag{223}$$

This run visits $n + 1$ states. But $\mathfrak{A}$ has at most n states, so there is a state, which has been visited twice. There are i and j such that $i < j$ and $q_i = q_j$. Then put $\vec{u} := x_0x_1 \cdots x_{i-1}$, $\vec{v} := x_ix_{i+1} \cdots x_{j-1}$ and $\vec{w} := x_jx_{j+1} \cdots x_{n-1}$. We claim that there

exists an accepting run for every string of the form $\vec{u}\vec{v}^{n}\vec{w}$. For example,

(224) $q_0 \xrightarrow{\vec{u}} q_i = q_j \xrightarrow{\vec{w}} q_n$

(225) $q_0 \xrightarrow{\vec{u}} q_i \xrightarrow{\vec{v}} q_j = q_i \xrightarrow{\vec{w}} x_n$

(226) $q_0 \xrightarrow{\vec{u}} q_i \xrightarrow{\vec{v}} q_j = q_i \xrightarrow{\vec{v}} q_j \xrightarrow{\vec{w}} x_n$

(227) $q_0 \xrightarrow{\vec{u}} q_i \xrightarrow{\vec{v}} q_j = q_i \xrightarrow{\vec{v}} q_j = q_i \xrightarrow{\vec{v}} q_j = q_i \xrightarrow{\vec{w}} x_n$

There are examples of this kind in natural languages. An amusing example is from Steven Pinker. In the days of the arms race one produced not only `missiles` but also `anti missile missile`, to be used against missiles; and then `anti anti missile missile missiles` to attach anti missile missiles. And to attack those, one needed `anti anti anti missile missile missile missiles`. And so on. The general recipe for these expressions is as follows:

(228) $(\texttt{anti}^\frown\Box)^n(\texttt{missile}^\frown\Box)^{n\frown}\texttt{missile}$

We shall show that this is not a regular language. Suppose it is regular. Then there is a k satisfying the conditions above. Now take the word $(\texttt{anti}^\frown\Box)^k(\texttt{missile}^\frown\Box)^{k\frown}\texttt{missile}$. There must be a subword of nonempty length that can be omitted or repeated without punishment. No such word exists: let us break the original string into a prefix $(\texttt{anti}^\frown\Box)^k$ of length $5k$ and a suffix $(\texttt{missile}^\frown\Box)^{k\frown}\texttt{missile}$ of length $8(k+1)-1$. The entire string has length $13k + 7$. The string we take out must therefore have length $13n$. We assume for simplicity that $n = 1$; the general argument is similar. It is easy to see that it must contain the letters of `anti missile`. Suppose we decompose the original string as follows:

(229) $\vec{u} = (\texttt{anti}^\frown\Box)^{k-1}, \vec{v} = \texttt{anti missile}^\frown\Box,$
$\vec{w} = (\texttt{missile}^\frown\Box)^{k-1\frown}\texttt{missile}$

Then $\vec{u}\vec{w}$ is of the required form. Unfortunately,

(230) $\vec{u}\vec{v}^2\vec{w} = (\texttt{anti}^\frown\Box)^{k-n\frown}\texttt{anti missile anti missile}^\frown\Box^\frown$
$(\texttt{missile}^\frown\Box)^{k-2\frown}\texttt{missile} \notin L$

Similarly for any other attempt to divide the string.

This proof can be simplified using another result. Consider a map $\mu : A \to B^*$, which assigns to each letter $a \in A$ a string (possibly empty) of letters from B. We extend μ to strings as follows.

$$\overline{\mu}(x_0x_1x_2\cdots x_{n-1}) = \mu(x_0)\mu(x_1)\mu(x_2)\cdots\mu(x_{n-1}) \tag{231}$$

For example, let $B = \{\mathtt{c}, \mathtt{d}\}$, and $\mu(\mathtt{c}) := \mathtt{anti}\hat{}\Box$ and $\mu(\mathtt{d}) := \mathtt{missile}\hat{}\Box$. Then

$$\overline{\mu}(\mathtt{d}) = \mathtt{missile}\hat{}\Box \tag{232}$$

$$\overline{\mu}(\mathtt{cdd}) = \mathtt{anti\ missile\ missile}\hat{}\Box \tag{233}$$

$$\overline{\mu}(\mathtt{ccddd}) = \mathtt{anti\ anti\ missile\ missile\ missile}\hat{}\Box \tag{234}$$

So if $M = \{\mathtt{c}^k\mathtt{d}^{k+1} : k \in \mathbb{N}\}$ then the language above is the $\overline{\mu}$–image of M (modulo the blank at the end).

Theorem 36 *Let $\mu : A \to B^*$ and $L \subseteq A^*$ be a regular language. Then the set $\{\overline{\mu}(\vec{v}) : \vec{v} \in L\} \subseteq B^*$ also is regular.*

However, we can also do the following: let ν be the map

$$\mathtt{a} \mapsto \mathtt{c}, \mathtt{n} \mapsto \varepsilon, \mathtt{t} \mapsto \varepsilon, \mathtt{i} \mapsto \varepsilon, \mathtt{m} \mapsto \mathtt{d}, \mathtt{s} \mapsto \varepsilon, \mathtt{l} \mapsto \varepsilon, \mathtt{e} \mapsto \varepsilon, \Box \mapsto \varepsilon \tag{235}$$

Then

$$\overline{\nu}(\mathtt{anti}) = \mathtt{c}, \overline{\nu}(\mathtt{missile}) = \mathtt{d}, \overline{\nu}(\Box) = \varepsilon \tag{236}$$

So, M is also the image of L under $\overline{\nu}$. Now, to disprove that L is regular it is enough to show that M is not regular. The proof is similar. Choose a number k. We show that the conditions are not met for this k. And since it is arbitrary, the condition is not met for any number. We take the string $\mathtt{c}^k\mathtt{d}^{k+1}$. We try to decompose it into $\vec{u}\vec{v}\vec{w}$ such that $\vec{u}\vec{v}^j\vec{w} \in M$ for any j. Three cases are to be considered. (Case a) $\vec{v} = \mathtt{c}^p$ for some p (which must be > 0):

$$\mathtt{cc}\cdots\mathtt{c} \bullet \mathtt{c}\cdots\mathtt{c} \bullet \mathtt{c}\cdots\mathtt{cdd}\cdots\mathtt{d} \tag{237}$$

Then $\vec{u}\vec{w} = \mathtt{c}^{k-p}\mathtt{d}^{k+1}$, which is not in M. (Case b) $\vec{v} = \mathtt{d}^p$ for some $p > 0$. Similarly. (Case c) $\vec{v} = \mathtt{c}^p\mathtt{d}^q$, with $p, q > 0$.

$$\mathtt{cc}\cdots\mathtt{c} \bullet \mathtt{c}\cdots\mathtt{cd}\cdots\mathtt{d} \bullet \mathtt{dd}\cdots\mathtt{d} \tag{238}$$

Then $\vec{u}\vec{v}^2\vec{w}$ contains a substring dc, so it is not in M. Contradiction in all cases. Hence, k does not satisfy the conditions.

A third theorem is this: if L and M are regular, so is $L \cap M$. This follows from Theorem **??**. For if L and M are regular, so is $A^* - L$, and $A^* - M$. Then so is $(A^* - L) \cup (A^* - M)$ and, again by Theorem **??**,

$$A^* - ((A^* - L) \cup (A^* - M)) = L \cap M \tag{239}$$

This can be used in the following example. Infinitives of German are stacked inside each other as follows:

(240) `Maria sagte, dass Hans die Kinder spielen ließ.`

(241) `Maria sagte, dass Hans Peter die Kinder spielen lassen ließ.`

(242) `Maria sagte, dass Hans Peter Peter die Kinder spielen lassen lassen ließ.`

(243) `Maria sagte, dass Hans Peter Peter Peter die Kinder spielen lassen lassen lassen ließ.`

Call the language of these strings H. Although it is from a certain point on hard to follow what the sentences mean, they are grammatically impeccable. We shall this language is not regular. From this we shall deduce that German as whole is not regular; namely, we claim that H is the intersection of German with the following language

(244) `Maria sagte, dass Hans`$^\frown(\square^\frown$`Peter`$)^{k\frown}\square^\frown$`die Kinder spielen`$^\frown(\square^\frown$`lassen`$)^{k\frown}\square^\frown$`ließ.`

Hence if German is regular, so is H. But H is not regular. The argument is similar to the ones we have given above.

Now, there is a similar property of context free languages, known as the **Pumping Lemma** or **Ogden's Lemma**. I shall give only the weakest form of it.

Theorem 37 (Pumping Lemma) *Suppose that L is a context free language. Then there is a number $k \geq 0$ such that every string $\vec{x} \in L$ of length at least k possesses*

a decomposition $\vec{x} = \vec{u}\vec{v}\vec{w}\vec{y}\vec{z}$ with $\vec{v} \neq \varepsilon$ or $\vec{y} \neq \varepsilon$ such that for every $j \in \mathbb{N}$ (which can be zero):

(245) $\vec{u}\vec{v}^j\vec{w}\vec{y}^j\vec{z} \in L$

The condition $\vec{v} \neq \varepsilon$ or $\vec{y} \neq \varepsilon$ must be put in, otherwise the claim is trivial: every language has the property for $k = 0$. Simply put $\vec{u} = \vec{x}$, $\vec{v} = \vec{w} = \vec{y} = \vec{z} = \varepsilon$. Notice however that $\vec{u}$, $\vec{w}$ and $\vec{z}$ may be empty.

I shall briefly indicate why this theorem holds. Suppose that $\vec{x}$ is very large (for example: let π be the maximum length of the right hand side of a production and $\nu = |N|$ the number of nonterminals; then put $k > \pi^{\nu+1}$). Then any tree for $\vec{x}$ contains a branch leading from the top to the bottom and has length $> \nu + 1$. Along this branch you will find that some nonterminal label, say A, must occur twice. This means that the string is cut up in this way:

(246) $x_0x_1 \cdots x_{i-1} \circ x_ix_{i+1} \cdots x_{j-1} \bullet x_j \cdots x_{\ell-1} \bullet x_\ell \cdots x_{m-1} \circ x_m \cdots x_{k-1}$

where the pair of $\bullet$ encloses a substring of label A and the $\circ$ encloses another one. So,

(247) $x_ix_{i+1} \cdots x_{m-1} \qquad x_j \cdots x_{\ell-1}$

have that same nonterminal label. Now we define the following substrings:

(248) $\vec{u} := x_0x_1 \cdots x_{i-1}, \vec{v} := x_i \cdots x_{j-1}, \vec{w} := x_j \cdots x_{\ell-1},$
$\vec{y} := x_\ell \cdots x_{m-1}, \vec{z} := x_m \cdots x_{k-1}$

Then one can easily show that $\vec{u}\vec{v}^j\vec{w}\vec{y}^j\vec{z} \in L$ for every j. We mention also the following theorems.

Theorem 38 *Let L be a context free language over A and M a regular language. Then $L \cap M$ is context free. Also, let $\mu : A \to B^*$ be a function. Then the image of L under $\overline{\mu}$ is context free as well.*

I omit the proofs. Instead we shall see how the theorems can be used to show that certain languages are not context free. A popular example is $\{\mathsf{a}^n\mathsf{b}^n\mathsf{c}^n : n \in \mathbb{N}\}$. Suppose it is context free. Then we should be able to name a $k \in \mathbb{N}$ such that for all strings $\vec{x}$ of length at least k a decomposition of the kind above exists. Now,

we take $\vec{x} = \mathsf{a}^k\mathsf{b}^b\mathsf{c}^k$. We shall have to find appropriate $\vec{v}$ and $\vec{y}$. First, it is easy to see that $\vec{v}\vec{y}$ must contain the same number of a, b and c (can you see why?). The product is nonempty, so at least one a, b and c must occur. We show that $\vec{u}\vec{v}^2\vec{w}\vec{y}^2\vec{z} \notin L$. Suppose, $\vec{v}$ contains a and b. Then $\vec{v}^2$ contains a b before an a. Contradiction. Likewise, $\vec{v}$ cannot contain a and c. So, $\vec{v}$ contains only a. Now $\vec{y}$ contains b and c. But then $\vec{y}^2$ contains a c before a b. Contradiction. Hence, $\vec{u}\vec{v}^2\vec{w}\vec{y}^2\vec{z} \notin L$.

Now, the Pumping Lemma is not an exact characterization. Here is a language that satisfies the test but is not context free:

(249) $C := \{\vec{x}\vec{x} : \vec{x} \in A^*\}$

This is known as the copy language. We shall leave it as an assignment to show that C fulfills the conclusion of the pumping lemma. We concentrate on showing that it is not context free. The argument is a bit involved. Basically, we take a string as in the pumping lemma and fix a decomposition so that $\vec{u}\vec{v}^j\vec{w}\vec{y}^j\vec{z} \in C$ for every j. The idea is now that no matter what string $y_0y_1 \cdots y_{p-1}$ we are given, if it contains a constituent A:

(250) $y_0g_1 \cdots y_{p-1} \bullet y_p \cdots y_{q-1} \bullet y_j \cdots y_{q-1}$

then we can insert the pair $\vec{v}, \vec{y}$ like this:

(251) $y_0y_1 \cdots y_{p-1} \bullet x_p \cdots x_{j-1} \circ y_i \cdots y_{j-1} \bullet x_\ell \cdots x_{m-1}y_p \cdots y_{q-1}$

Now one can show that since there are a limited number of nonterminals we are bound to find a string which contains such a constituent where inserting the strings is inappropriate.

Natural languages do contain a certain amount of copying. For example, Malay (or Indonesian) forms the plural of a noun by reduplicating it. Chinese has been argued to employ copying in yes-no questions.